EL LIBRO NEGRO del Programador_

Por Rafael Gómez Blanes



Cómo conseguir una carrera de éxito desarrollando software y cómo evitar los errores habituales_

El siglo XXI es el de la sociedad de la información y las nuevas tecnologías: todo ello no sería posible sin la enorme industria del software que le sirve de base.

No obstante, los desarrolladores de software no aprovechan todas las oportunidades para desempeñar una carrera profesional de éxito, cometiendo siempre los mismos errores una y otra vez. Un buen proyecto software tiene que ver con habilidades creativas y artísticas más que aquellas necesariamente técnicas.

El Libro Negro del Programador muestra qué distingue a un programador neófito de quien actúa y trabaja profesionalmente.

En la era del emprendimiento y de la nueva economía, el desarrollo profesional de software es su pilar fundamental. Si como programador quieres llegar a ser no sólo bueno sino profesional, no puedes dejar de conocer las gemas de sabiduría que contiene *El Libro Negro del Programador*.



Rafael Gómez Blanes

El libro negro del programador

Cómo conseguir una carrera de éxito desarrollando software y cómo evitar los errores habituales

ePub r1.0
Titivillus 07.07.17

Título original: *El libro negro del programador*
Rafael Gómez Blanes, 2004
Diseño e ilustraciones: Francisco Soria

Editor digital: Titivillus
ePub base r1.2



¿TE HAS SENTIDO ASÍ ALGUNA VEZ?

El Libro Negro del Programador te va a mostrar qué distingue a un buen profesional del software de otro que «juega» con la tecnología sin conseguir resultados importantes: en el nuevo paradigma laboral trabajaremos principalmente con la máxima productividad y por resultados, para lo que es imprescindible cultivar buenos y productivos hábitos de trabajo.

Introducción

«Emplea tu tiempo cultivándote a través de los escritos de otros, así ganarás fácilmente lo que para nosotros ha sido una dura tarea.»

(Sócrates)

Estamos viviendo momentos apasionantes: un nuevo paradigma económico ha venido para quedarse, podemos contratar casi cualquier servicio en cualquier parte del mundo, podemos diseñar aquí y fabricar por encargo a miles de kilómetros de distancia y del mismo modo lo que fabricas y produces en tu compañía puede ser comercializado en muchas partes del mundo. Profesiones que antes no existían ahora están a la orden del día, una rotación laboral alta está dejando obsoletos los puestos de trabajo «para toda la vida»; hay quien define todo este escenario como crisis, otros como un paradigma y orden nuevo cargado de oportunidades. Hay opiniones entre las que elegir.

Sin embargo, todo esto ha sido posible gracias al despliegue global de redes de comunicaciones que nos presentan a «golpe de clic» oportunidades que antes eran inimaginables. Hay quien dice que todo lo que estamos viviendo es una consecuencia natural de la implantación y uso de Internet.

El protagonista de esta revolución silenciosa, es, en esencia, el desarrollo de software: todo a nuestro alrededor está informatizado y, sin embargo, la profesión informática sufre de cierta incomprensión por el resto de la sociedad: a los desarrolladores de software se les encasilla fácilmente, se les asocia frívolamente con una imagen de «juventud», de trabajo en garajes destartalados cuando la realidad es que un buen proyecto sólo sale adelante con un alto grado de especialización y experiencia.

¿Somos conscientes de la enorme facilidad que tenemos hoy para emprender y poner en marcha ideas innovadoras desde el salón de casa?, ¿de que las barreras de entrada en muchos negocios han caído estrepitosamente? ¿Nos damos cuenta de que toda esta revolución la está protagonizando de una u otra manera el software que se ejecuta en millones de servidores por todo el mundo?

Recientemente se ha publicado un estudio según el cual en EEUU ya es mayor la masa laboral que trabaja «desarrollando aplicaciones» que el número de trabajadores dedicados a la agricultura, sorprendente, ¿no? Igualmente, no paro de leer, en estos mismos días, que el desarrollo de software será una de las profesiones más demandadas en el futuro próximo, lo está siendo ya, pero esta vez en forma de analistas web, SEO (*search engine optimization*), expertos en *big data*, *community manager*, etc. Es una profesión de futuro, efectivamente (de presente lo es ya), pero me pregunto cuáles serán las características que distinguirán a un buen profesional de otros cuyo trabajo deja que desear.

No obstante, la realidad es a veces totalmente frustrante: el poder de la tecnología se desvanecería si no tuviera el soporte de miles de desarrolladores de software afanándose día a día en que esta no colapse, haciendo parches (*hotfixes*) de soluciones cutres, manteniendo webs horrorosas cuyas compañías quieren rentabilizar al máximo o desarrollando *apps* para obtener algunos ingresos pasivos… Es una profesión poliédrica, multidisciplinar en la que dentro de la misma te puedes dedicar a hacer multitud de cosas extraordinariamente diferentes. Pero todas, absolutamente todas, se hacen bien, con profesionalidad, si somos capaces de rodearnos de buenos hábitos y de comprender la naturaleza frágil, artística y creativa del software.

Y luego viene la cadena de mando…, cuando el buen hacer técnico, los años de experiencia desarrollando arquitecturas software eficientes y correctas se tiran a la basura porque la única manera de prosperar (ganar más dinero, lógicamente), es pasar «al siguiente nivel» como manager y comenzar a trabajar de lo que un informático desarrollador de software apenas tiene idea, para lo que apenas o nada ha sido formado académicamente: gestionar un equipo de trabajo. ¿Pero qué tendrá que ver poner en marcha una arquitectura SOA o una metodología *extreme* con gestionar el tiempo y el

trabajo de un equipo de personas, las cuales en su mayoría te miran con aprensión porque ven que para ellos cobrar algo más es hacer lo que tú acabas de conseguir?

En estos lamentables casos, la mentalidad empresarial dominante que te premia con asumir más «responsabilidad» lo que consigue es perder un magnífico técnico y ganar un pésimo gestor. A mí esto me ha pasado y es comidilla habitual en el sector. Y, sin embargo, ¡cómo desprestigia esta dinámica e inercia el hecho de programar!, ya que, si te pagan más por «gestionar y coordinar» será que lo otro es menos importante… ¿Conocéis quien tenga mejor retribución económica programando que gestionando grupos de trabajo o proyectos?

Entonces, ¿qué futuro laboral se puede esperar de un amante de la tecnología, de un apasionado empleado que pasa tantísimas horas delante de un ordenador depurando ese error sutil durante horas o de quien se esfuerza por generar el mejor software mantenible y ligero pero que ni su jefe ni su cliente van a valorar porque a estos lo único que les interesa medir es el tiempo y el coste a corto plazo (cosa lógica, por otra parte)?

El desarrollo de software es una profesión de presente y de enorme futuro, y, sin embargo, no sabemos aprovechar al máximo este nuevo paradigma económico cargado de oportunidades: caemos siempre en los mismos errores, una y otra vez, sin darnos cuenta de que la única ventaja competitiva que tendremos será la de la calidad e innovación.

Por eso os presento este libro, donde resumo mi más o menos dilatada experiencia laboral y de *freelance*, llena de desaciertos, errores y decepciones pero también llena de éxitos profesionales que me han permitido seguir amando mi actividad diaria. En *El Libro Negro del Programador* encontrarás los errores más habituales que solemos cometer los desarrolladores profesionales de software y, sobre todo, veremos cómo evitarlos. Lo más interesante de esto es que la mayoría de esos errores, tropiezos y problemas que terminan haciendo fracasar un proyecto, no son de carácter técnico.

A diferencia de otras profesiones, el factor vocacional en el desarrollo de software es importante: los mejores desarrolladores son los que verdaderamente disfrutan más con lo que hacen. Y es que, ¿no deberíamos aspirar a amar todo lo que hacemos?, ¿o es que nos tenemos que conformar

con pasar ocho o más horas al día vendiendo nuestro tiempo para pagar las facturas? Vamos hacia una economía en la que el tiempo que pasamos en un trabajo será más y más irrelevante: lo más importante serán los resultados y objetivos cumplidos; nos pagarán por proyectos y el trabajo «en nómina» será cada vez más escaso, sólo para el personal estructural necesario para la supervivencia de la compañía.

Aquí va el apasionante viaje de un ingeniero informático obsesionado por mostrar la especial idiosincrasia de nuestra profesión y, sobre todo, enseñar que hay muchas circunstancias no técnicas que «rodean» nuestra actividad y que determinan nuestro éxito o fracaso.

Quiero mostrar también en qué términos se debe considerar un software bien hecho como parte de un proceso más «artístico» que técnico; para los que aún no os habéis dado cuenta, los desarrolladores de software tenemos más de artistas que de frías mentes matemáticas. Creamos artefactos de la nada que tienen que evolucionar y ser mantenidos durante mucho tiempo.

Simplicidad, diseños emergentes, deuda técnica, los hábitos del buen desarrollador, trabajar en equipo, refactorings, desarrollo de software testeable, la rentabilidad metodológica, horas o productividad, buena organización, abstracción bajo principios, disciplina de trabajo, reutilización y desacoplamiento de módulos y librerías, (y un larguísimo etcétera) son los ingredientes y el vocabulario habitual de este libro con el que espero crear, formar y motivar a más y mejores profesionales.

Este libro está escrito por un desarrollador de software por y para otros programadores y desarrolladores. Creo firmemente que el mejor currículum que podemos mostrar es el de un buen trabajo realizado con la máxima calidad y sobre todo de enorme utilidad. De nosotros depende, en cierta medida, rodearnos de las «condiciones» adecuadas para conseguir ese objetivo.

Excepciones lingüísticas

A lo largo del libro se usan de manera natural algunos anglicismos y palabras que aunque no están plenamente reconocidos en ninguna de las variantes del español, forman parte del acervo cultural de cualquier desarrollador de software. En este sentido, uso libremente los términos depurar, release, mantenibilidad, refactoring, framework, bug, manager, tester, etc.

En ciertos contextos se diferencian los términos programador, desarrollador de software, hacker, analista, etc. En *El Libro Negro del Programador* entendemos por programador o desarrollador a quien escribe «líneas de código» en general.

Manifiesto

Tienes en tus manos un libro que va a mejorar radicalmente cómo actúas como desarrollador de software: los capítulos que siguen a continuación te van a mostrar el «terreno» que tienes que cultivar para hacer proyectos con la máxima calidad y mejorar sustancialmente tu carrera como programador profesional. La calidad es, en definitiva, el único currículum que podemos aportar sobre nuestro trabajo.

La diferencia entre ser un programador de baja clase, quizá frustrado con tu profesión o bien ser un desarrollador de software con muchos éxitos a tu espalda es una cuestión de sencillos hábitos de trabajo pero, sobre todo, saber ver que el software tiene su propia idiosincrasia creativa y artística, no sólo técnica; de aquí el siguiente manifiesto de *El Libro Negro del Programador*, toda una declaración de principios.

El desarrollo de software profesional tiene tantos aspectos artísticos y creativos como técnicos.

Recibimos una formación de años para hacernos con multitud de conocimientos que más tarde difícilmente vamos a poder aplicar.

Desarrollamos proyectos en compañías que desconocen la naturaleza frágil y evanescente de un producto software bien desarrollado, aplicando criterios de producción industrial.

Nos gusta investigar (lo cual es bueno) y alardear (lo cual no lo es) del conocimiento de toda tecnología emergente que va surgiendo. En cierto modo somos víctimas de un diletantismo un poco infantil.

Mostramos resultados espectaculares cuando internamente el nivel de fragilidad de una aplicación te obligará a tirarla al cabo de meses por su inmantenibilidad.

Olvidamos a menudo que cualquier éxito en software (y en otras profesiones) es una mezcla de tecnología, talento y metodología. Nada medianamente complejo se puede realizar sin un guion previo o marco de trabajo en el que hacer las cosas ordenadamente (metodología).

Por lo general, responsables, jefes y capas de decisión no tienen una idea clara de la naturaleza creativa y artística del desarrollo de software.

En nuestra etapa académica no se nos enseña ni a trabajar en equipo ni que el éxito de un proyecto software nace de la buena colaboración de entre todos sus miembros.

Hay a quienes les gusta ir de gurús, llenos de soberbia cuando les reconocen como tales, pero cuyo individualismo les impide trabajar con éxito dentro de un equipo.

Confundimos cumplir las fechas con entregar un producto de calidad.

A veces nos gusta ser imprescindibles y hacer las cosas enrevesadas para apropiarnos de una parcela de poder y sentirnos imprescindibles (convirtiéndonos en esclavos de nosotros mismos).

No aprovechamos lo suficiente la enorme capacidad emprendedora que podemos tener aplicando nuestro conocimiento de la tecnología en una economía que se mueve hacia lo digital: el «talentismo» distinguirá en el futuro a los profesionales de los que no lo son.

Necesitamos, por tanto, un libro que recoja estas reflexiones y nos enseñe cómo abordar todos estos aspectos de una profesión que ni está suficientemente valorada socialmente ni suficientemente recompensada económicamente.

¿Te has sentido identificado con alguno de los puntos del manifiesto? Veremos a lo largo de los capítulos de *El Libro Negro del Programador* cómo combatir los malos vicios de nuestra profesión, esa cultura empresarial que en ocasiones trata el software sin tener en cuenta su naturaleza para su correcto desarrollo y gestión y, lo mejor de todo, veremos cuáles son los verdaderos hábitos que un buen programador tiene que cultivar para hacer extraordinariamente bien su trabajo.

Sostengo que el desarrollo de software tiene más características de arte que de ciencia técnica, construimos más que calculamos, creamos más que ensamblamos.

Desarrollo, pruebas, refactoring (todo va en el mismo paquete)

«Creamos software de producción, lo probamos con tests que podemos automatizar y nos paramos un segundo a preguntarnos ¿puedo mejorar esto o aquello? Estos son los hábitos que debemos incorporar a nuestro arte de programar, la deuda de no incluirlos a nuestro ADN de programadores profesionales es demasiado alta.»

En ocasiones un proyecto software termina siendo un jardín abandonado lleno de malas hierbas. Dispongo de una casa con una zona ajardinada más o menos grande; es sorprendente ver cómo la naturaleza prolifera por sí misma, a su propio ritmo, si se la deja crecer a su gusto. En pocas semanas unas enormes malas hierbas con hojas puntiagudas comenzarán a invadir poco a poco el césped; pequeñas y bonitas plantitas cuya especie aún no he sabido identificar aparecen por aquí y por allá; en poco tiempo, estas inofensivas malas hierbas habrán alcanzado un tamaño considerable. Trozos dispersos de grama crecen a su antojo y forman enormes melenas que de no actuar a tiempo te obligarán a usar una azada grande para eliminarlas, dejando un agujero en la tierra. Si en meses no haces absolutamente nada, lo que antes era un trozo de parcela plano comienza a cobrar una orografía vegetal frondosa; si sigue pasando el tiempo, menos de lo que imaginamos, por donde antes podían jugar mis hijas se ha convertido ya en una zona completamente intransitable.

Es entonces cuando la situación no puede esperar más y con más obligación que ganas, decides un fin de semana pasarte sábado y domingo limpiando el desmadre vegetal antes de que los vecinos confundan tu casa

con la de «Jumanji» (película que por cierto me encanta). Con enorme esfuerzo y tedio, tienes que emplearte a fondo para limpiar malas hierbas desbrozando por todos lados. El trabajo es agotador y sólo te sirve para volver a la situación de partida de meses atrás; si no decides usar unos herbicidas potentes, al tiempo tendrás exactamente la misma situación.

Pues bien, de ese modo evoluciona a menudo el desarrollo de un proyecto software. Si desde mucho antes me hubiera dedicado un poco a esa agradable afición que se llama jardinería doméstica, la situación no se habría deteriorado tanto, no habría llegado a ese fin de semana tan frustrado y la parcela se habría conservado relativamente limpia de malas hierbas con mucho menor esfuerzo. Y es que en muchas ocasiones no sabemos valorar el «coste oculto» que tiene el no hacer algo. Lo que a pequeñas dosis era algo abordable y hasta agradable (relajados ejercicios de jardinería) se puede convertir en una auténtica pesadilla meses después.

La cuestión aquí es si hemos incorporado o no algunos hábitos que evitan este tipo de situaciones.

Como desarrolladores de software nos pasa exactamente lo mismo: no conocemos (o no nos han enseñado) lo suficientemente bien lo que deben ser nuestros buenos hábitos para el día a día.

Cuando comenzamos una nueva solución nos gusta empezar a teclear líneas y líneas de código por la ansiedad de tener (y a lo mejor mostrar) algo funcionando cuanto antes, o por el contrario nos encontramos de golpe con todo el trabajo acumulado a medida que se acercan las fechas previstas de entrega.

Avanzamos generando software «de producción» sin asegurarnos lo suficientemente bien de que aquello que vamos construyendo tiene que ir siendo probado con la garantía correcta. ¿Cuántos no nos hemos hecho el típico programa de consola fuera de la solución para probar algo en concreto? Este es uno de nuestros mayores vicios: creamos software de producción sin software que lo pruebe o respalde y nos muestre a nosotros mismos que funciona. He pasado mucho tiempo analizando por qué esto ocurre así, por qué cuesta tanto trabajo que el equipo con el que colaboras no abandone el desarrollo de pruebas en ninguna fase del proyecto y lo que no es poco, que la calidad de esas pruebas sea la adecuada. La teoría nos muestra que un buen

equipo de desarrollo cuenta con sus propios testers quienes se encargan de escribir las pruebas para respaldar el trabajo del resto de desarrolladores; no obstante, la realidad es bien distinta: ¿en cuántos proyectos habéis trabajado con la suerte de contar con roles específicos para este propósito?

La razón principal para obviar el desarrollo de pruebas a medida que hacemos software de producción es porque nos hemos acostumbrado a hacer software «no enfocado a pruebas»: efectivamente, para generar tests que se puedan reproducir automáticamente una y otra vez se requiere que el software que desarrollas sea «testeable». Para ello, la manera en que abstraemos un problema y lo convertimos en piezas de código tiene que cambiar necesariamente. Una aplicación puede funcionar bien pero puede no estar escrita de modo que se puedan crear pruebas automáticas para comprobar su comportamiento.

La naturaleza y diseño de un software testeable es completamente distinta de un código que se ha hecho sin el hábito continuo de respaldarlo con pruebas.

La segunda razón (pero no menos importante) es porque no percibimos el «coste oculto» que tiene no implementar esas pruebas: dejamos así que las malas hierbas sigan creciendo semana tras semana hasta que llega el momento en que comienzas a criar animales exóticos en la selva en la que se ha convertido tu jardín o tienes que contratar a alguien para que haga una limpieza a fondo (a un coste infinitamente mayor que si de vez en cuando hubiéramos hecho unos trabajos simples de limpieza en el jardín). Esto traducido a un proyecto software es lo mismo que tirarlo a la basura tarde o temprano.

La tercera razón (y creedme que es la que menos me gusta) es porque nadie nos obliga: si no convives con un grupo de trabajo mentalizado en que se debe hacer software de producción soportado por pruebas, tú no vas a convertirte en el «rarito» que avance de esa manera, a no ser que tu jefe o tu manager te obligue explícitamente a hacerlo; en este caso, ¿cómo controla tu manager la calidad o cobertura de las pruebas que haces sin tener que dedicarse a ello al 100%?

Aquí presento tres razones que podemos resumir en una mucho más sencilla: no implementamos tests porque no tenemos el hábito de

desarrollarlos. Es exactamente lo mismo que cuando no te lavas los dientes después del almuerzo: se te queda un cuerpo incómodo y extraño hasta que te los cepillas, al menos a mí me pasa, no podría conciliar el sueño si antes no me he cepillado los dientes. Igualmente, yo no podría sentirme a gusto si no refuerzo el código que escribo suficientemente con sus correspondientes pruebas.

El error se intensifica cuando comenzamos a tener menos tiempo para todo el trabajo que nos queda por delante: el estrés y las prisas obstaculizan el avance del trabajo e impiden madurar las decisiones de diseño, probar la mejor solución y, cómo no, la creación de buenos tests unitarios o de integración.

Avanzar en los desarrollos con pruebas de calidad lo debemos incorporar a nuestro ADN de desarrolladores; hoy día no es excusa el tener que usar terceras herramientas o extraños frameworks de pruebas porque estos son suficientemente conocidos y maduros cuando no están integrados en el mismo IDE de desarrollo.

El hábito aquí es la clave y el profundo conocimiento de que si no apoyas tu software con tests, el coste en tiempo, horas, calidad, euros y clientes insatisfechos será muchísimo mayor y puede que termine arruinando el proyecto.

Una vez leí una estadística sobre las marcas de automóviles para las que las facturas de reparación eran más bajas: coincidía con las marcas y modelos de los coches más fáciles de reparar. Esto que parece tan evidente ¿por qué no lo percibimos como programadores profesionales? Nuestro software será mejor (barato) si es fácil de reparar (depurar).

Nos encontramos de nuevo con un concepto bastante sutil: el diseño de una aplicación susceptible de ser probada mediante tests automáticos (unitarios, de integración, etc.) seguramente no tenga nada que ver con el de una aplicación cuyo diseño rígido impide generar pruebas. En este último caso, las únicas pruebas posibles son las manuales, pero ¿verdad que no queremos realizar trabajos repetitivos sino concentrar nuestros esfuerzos en la creatividad de resolver problemas?

Lo que se nos escapa a menudo es que un software testeable debe tener una estructura, diseño y concepción ¡testeable!; esto es exactamente lo mismo

que las diferencia entre modelos de vehículos que son fáciles de reparar de los que son un auténtico martirio sustituirles tan siquiera una lámpara de cruce.

Sólo podemos crear software testeable si abstraemos la solución lo suficientemente bien como para luego permitir el desarrollo de las pruebas correspondientes. Esto requiere cierto cambio de mentalidad a la hora de enfocar los problemas aunque también es una cuestión de implantar ciertos principios de diseño bien conocidos, tales como S.O.L.I.D, KISS (*keep it simple, stupid!*), DRY (*don't repeat yourself*) y un larguísimo etcétera. Esto necesita de práctica pero sobre todo de querer y tener el firme propósito de «re-aprender» nuestro oficio para permitir avanzar nuestro código productivo con tests.

Hace unos años emprendí una mega reforma de mi casa que mi pareja y yo estuvimos preparando durante mucho tiempo. Cambiamos la cocina de lugar, buscamos la mejor distribución para las cosas y analizamos todos los detalles para que la enorme inversión de dinero (y esfuerzo) que nos suponía la reforma nos diera exactamente el tipo de hogar que estábamos buscando. Pues bien, a pesar de haberlo planeado todo con extremo detalle, al año nos dimos cuenta de que pusimos una mosquitera en una ventana interior que realmente no la necesitaba (…), el sacar del lavaplatos la vajilla se convierte en una maratón porque el mueble donde se guardan platos y vasos está en el otro extremo de la cocina, una de las ventanas de esta da justo a un enorme árbol con un gran follaje que impide la entrada de luz natural suficiente.

Una vez que te das cuenta de estos errores no previstos, no puedes volver atrás sin que esto suponga otro quebradero de cabeza o un quebranto económico: por tanto te quedas como estás y siempre te consuela eso de decir «qué pena que no nos diéramos cuenta de…».

Sin embargo, en software tenemos la habilidad y capacidad de destruir, reconstruir y modificar lo que desarrollamos a nuestro antojo: es más, esta modificación para la mejora no es un paso atrás, sino todo un paso adelante si ello supone mejorar el diseño o implementación de la solución. Es una inversión aunque en ocasiones nos cueste mucho eliminar o modificar algo que nos costó horas o días desarrollar. A esta técnica la llamamos «refactoring» y no es más que buscar cómo mejorar o simplificar lo que ya

tenemos desarrollado.

Es todo un clásico el libro de Martin Fowler titulado «*Refactoring: improving the design of existing code*»; este libro me ha ayudado como ningún otro a mejorar en la calidad del código que escribo y debería estar en la estantería de cualquier programador profesional.

El resultado de refactorizar con sentido común nuestra solución es que esta va a tener un diseño más limpio, va a ser más testeable, tendrá menos duplicidades y en general resolverá los problemas de una manera más sencilla, facilitando a otro miembro del equipo su evolución, entre muchos otros argumentos a favor de refactorizar o no. Considero este tema de crucial importancia, tanto que hasta me atrevo a decir eso de que «programo, ergo refactorizo», una cosa tiene que ir con la otra. El desarrollo de pruebas nos va a indicar qué partes del sistema deben ser mejoradas. Y es que este es ni más ni menos que otro de los pilares del desarrollo de un software profesional y maduro: refactorizamos a medida que avanzamos generando cada vez una solución mejor.

Creamos software de producción, lo probamos con tests que podemos automatizar y nos paramos un segundo a preguntarnos ¿puedo mejorar esto o aquello? Estos son los hábitos que debemos incorporar a nuestro arte de programar, la deuda de no incluirlos a nuestro ADN de programadores profesionales es demasiado alta.

Cuando comento estos conceptos a menudo me sorprenden diciendo que el incorporar estos hábitos hacen que «tardemos más» en generar la solución: esto sólo se puede afirmar desde la más cándida inexperiencia. La deuda técnica de no hacerlo hará que más adelante tengamos una solución infinitamente más difícil de mantener, evolucionar y probar, volviendo a la relación de Pareto 20/80 que en su versión particular para el software significa que pasamos un 20% de nuestro tiempo escribiendo código productivo y un 80% ¡depurándolo!, cuando lo que perseguimos es precisamente invertir esa tendencia; ¿cómo?, creo que en este punto ha quedado suficientemente claro: apoyando nuestro software con pruebas automatizadas a medida que avanzamos y antes de dar un nuevo paso, refactorizando y simplificando como hábitos continuos. La solución irá generando un diseño cada vez mejor y ganando en calidad al tiempo que el

código ganará en limpieza y sencillez.

Puntos clave

- Es imprescindible avanzar en el código nuevo que incorporamos a nuestro proyecto respaldándolo con pruebas; estas deben ser de la misma calidad que el código de producción.
- El coste de no respaldar en mayor o menor medida nuestro proyecto con pruebas puede suponer que su tiempo de vida sea mucho menor: será más difícil y caro evolucionarlo y mantenerlo y seguramente llegue al cliente con errores no detectados.
- Antes de dar algo por finalizado como por ejemplo una nueva clase o una funcionalidad, nos debemos preguntar si hay algo que podamos hacer para mejorarlo o simplificarlo. Es sorprendente cuando nos damos cuenta que refactorizar en la mayoría de las ocasiones supone realmente poco esfuerzo.
- Si en nuestro equipo de trabajo aún no hay establecida una cultura sólida de creación de pruebas, debemos intentar establecerla: la calidad de lo generado será mejor y a la larga pasaremos menos tiempo detectando y corrigiendo errores cuando el proyecto pasa a producción.
- Las pruebas deben ser de calidad: éstas también deben ir siendo refactorizadas a medida que avanza el proyecto.
- Debemos hacernos continuamente preguntas del tipo ¿puedo simplificar este código de algún modo?, ¿puede ser entendido fácilmente por otra persona?
- Es indispensable para programar bien, conocer todas las técnicas necesarias para refactorizar y generar código limpio.

Qué es tener éxito en un proyecto software

«Un proyecto software no puede considerarse de éxito aunque se entregue a tiempo cuando el equipo que lo desarrolla trabaja bajo el síndrome del quemado, la hostilidad y hermetismo entre los desarrolladores son enormes y cuando muchos esperan la más mínima pifia del compañero de al lado para tapar sus propias miserias. Sutilmente, en un ambiente así, se desarrollará un código de pésima calidad, frágil y con tendencia a la corrupción de diseño. Necesariamente la solución desarrollada va a carecer de los pilares fundamentales de un buen software.»

Un software se puede entregar a tiempo y ser a la vez un absoluto y rotundo fracaso, así de sencillo y al mismo tiempo así de paradójico.

Muchas veces me he preguntado qué determina el éxito de un proyecto. Sorprendentemente no es entregarlo en los términos acordados en una fecha más o menos prevista, hay infinidad de factores sutiles y subyacentes que nos indican realmente el nivel de fracaso o éxito conseguido.

Hace unos años tuve la suerte (o mala suerte) de trabajar en un proyecto internacional que «en un principio» y visto desde el exterior, podía parecer verdaderamente apasionante. Últimas tecnologías, un equipo con experiencia, todo un año por delante con tiempo y todos los recursos necesarios. ¡Por fin había un grupo de testers! Creo recordar que éramos más de doce las personas involucradas.

Sin embargo, al cabo de diez meses era evidente que lo que comenzó con enormes perspectivas iba a terminar en un gran fiasco, un tremendo y monumental fracaso; entonces el problema ya no era si se llegaría a tiempo o

no, cosa evidente de que no iba a ser así, sino qué cabezas rodarían… Se entró entonces en una dinámica de «¡sálvese el que pueda!» Un fracaso en toda regla, sí señor, y, sin embargo, ese proyecto me enseñó tantísimas cosas que en realidad agradezco que fuese todo un fiasco: muchas de las habilidades y hábitos positivos que he aplicado desde entonces los aprendí precisamente después de cometer tantos errores (y de ver también cómo otros ponían su granito de arena en el asunto).

A la capacidad de aprender de los errores y reponerse del fracaso se llama «resiliencia» y dado que trabajamos en una disciplina en la que los mecanismos para cometer errores están tan a mano y son tan etéreos, un buen desarrollador de software debe tener esta capacidad sí o sí: aprendemos más de los errores que de los éxitos.

Estoy convencido de que el desarrollo de software es de las profesiones en las que más fracasos podemos acumular, por tanto, un buen ingeniero tiene que desarrollar una gran capacidad de resiliencia.

Partiendo de un prometedor proyecto, con recursos y suficiente tiempo por delante, se terminó con un equipo totalmente «quemado» (era evidente el síndrome de burnout), echando horas extras que por supuesto nadie iba a pagar, en un ambiente marciano de lo más hostil y donde ya no se hablaba, sino que se ladraba; las reuniones terminaban con un corolario de acusaciones personales y la aversión entre varios miembros clave del equipo crecía día a día. Los cotilleos, los rumores, los correos sarcásticos, el tú más y las puñaladas por la espalda estaban a la orden del día. Parece exagerado que lo diga así, pero os aseguro que la tensión se palpaba en el ambiente. Lógicamente este ambiente afectaba muy negativamente al proyecto y a la calidad de lo que en él se hacía.

Yo era uno de esos «miembros clave» del equipo (¡aunque ahora ya no estoy tan seguro!) de modo que sufría quizá más que el resto esta situación al darme cuenta claramente de que así era imposible llegar a los objetivos comunes del proyecto; éste no es más que eso: un objetivo común para el conjunto de personas que trabajan en él; por tanto, la sintonía del equipo redundará en mejores resultados.

El fracaso era más que evidente, aunque no lo era tanto la pérdida de «calidad» del código que se desarrollaba bajo el influjo de un ambiente tan

extremadamente hostil como aquel. Las capas de decisión, obsesionadas en aquel momento por llegar a tiempo a las fechas de entrega, estaban a años luz de entender esto. En cualquier caso, ¿se habría considerado un éxito si se hubiera llegado a tiempo? De haber sido así, el manager, el director y alguno más por encima se habrían dado por satisfechos, sin imaginar la herencia envenenada de lo entregado.

Un proyecto software no puede considerarse de éxito aunque se entregue a tiempo cuando el equipo que lo desarrolla trabaja bajo el síndrome del quemado, la hostilidad y hermetismo entre los desarrolladores son enormes y cuando muchos esperan la más mínima pifia del compañero de al lado para tapar sus propias miserias. Sutilmente, en un ambiente así, se desarrollará un código de pésima calidad, frágil y con tendencia a la corrupción de diseño. Necesariamente la solución desarrollada va a carecer de los pilares fundamentales de un buen software.

De este modo el camino a la «espaguetización» del código está servido y asegurado (permitidme esta referencia culinaria; el «espagueti software» es un concepto con el que todos deberíamos estar familiarizados).

Una situación similar seguramente pase en muchos contextos laborales, pero sucede que en software esta dinámica destructiva tiene consecuencias intangibles al principio y no tan fáciles de detectar a corto plazo, aunque manifiestamente evidentes a largo plazo.

El «arte de programar» requiere de un entorno altamente creativo, lúcido, tranquilo, optimista y positivo. En realidad, cuando desarrollamos una solución, una API, un módulo, una clase, un algoritmo, cualquier artefacto de sobra conocido por nosotros, estamos proyectando el estado mental de ese momento mientras escribimos líneas y líneas de código; diseñamos algo elegante cuando nuestra mente funciona y fluye con la suficiente tranquilidad y relajación.

No podemos comparar algo hecho con «amor» con algo que fabricamos esperando que el reloj marque el final de la jornada laboral, con prisas y sin ningún tipo de apego por lo que hacemos. No digo que el día de los enamorados nos acordemos de esas preciosas líneas de código… sino que tenemos que entender que sólo podemos brillar y hacer algo con excelencia si nos gusta, si nos apasiona.

Tampoco podemos esperar tener la suficiente inspiración para llegar a esa solución elegante y simple que necesitamos si nuestra mente está invadida por problemas y conflictos en el grupo de trabajo al tiempo que le damos codazos al de al lado (esto es figurativo, claro). Muchísimo menos vamos a tener el acicate y motivación necesarios para refactorizar, por ejemplo, eliminando esa duplicidad tan evidente, total, ¿para qué, si lo que quiero es que llegue las cinco para largarme? (espero que se capte la ironía).

Por tanto, aunque el proyecto hubiese sido entregado a tiempo y si los managers se hubieran colgado sus medallas por haber conseguido «apretar al equipo» (cuánto les gusta esta expresión a los responsables de medio pelo), lo entregado habría sido de una calidad pésima; las consecuencias de esto a medio plazo para un proyecto software son nefastas. Se entrega a tiempo, sí, pero se está dando una manzana envenenada.

Bien es sabido que ante un exceso de presión, nos vamos relajando en el desarrollo de las pruebas (si es que se hacen), vamos olvidando los numerosos «*to do…*» («por hacer», etiqueta habitual que incluimos para recordarnos que en ese punto concreto aún nos falta por terminar algo), comentarios que continuamente dejamos entre líneas de código y más nos acercamos al «happy path» cuando hay que probar algo. Esto último es una tendencia habitual cuando la presión nos acecha: el probar «el camino feliz», cuando vamos relajándonos en las pruebas y las pocas que hacemos precisamente son las que sabemos inconscientemente que menos problemas nos darán, las menos problemáticas.

Esta dinámica o inercia la sufre cualquier rol de un equipo de software: desarrollador, tester, etc.

El resultado es una solución extremadamente frágil, hecha con prisas y para salir del paso; en lugar de un rascacielos (que era lo que pretendemos siempre al inicio de un proyecto), terminamos entregando una cutre casita de adobe que cuando llueva se desintegrará hasta venirse abajo. En muchos casos y bajo estas circunstancias hemos entregado a tiempo algo que funciona, eso es evidente, porque hasta el más ingenuo de los managers se habrá cuidado de cerciorarse de que la cosa funciona antes de darla por cerrada o pasar a la siguiente etapa; entonces, ¿dónde está la herencia envenenada? Se habrá entregado bajo presión, con prisas y con un equipo

hastiado de gente quemada una solución extremadamente frágil que se romperá en cuanto se le exijan los más mínimos cambios, cuando la batería de pruebas de validación del cliente cambie lo más mínimo o en cuanto haya que evolucionar lo hecho hacia nuevas características no previstas en un principio. En un ambiente de trabajo hostil, sea cual sea la naturaleza de este, no se puede implantar ninguna de las características del software ágil y programación «extrema»: buenas prácticas y hábitos que aumentan en varios niveles la calidad de nuestro software y sobre todo nuestra productividad.

Es entonces cuando la cara inicialmente satisfecha de los managers y directores y jefes y… (¿por qué habrá siempre tanta cadena de mando?, ¿y tan larga?), esas caras de satisfacción, digo, se tornarán caras de preocupación, de turbación general, cuando vean posteriormente que los tiempos de adaptación, de mejoras o de corrección de errores son extremada y anormalmente largos. Se ha entregado, en definitiva, una solución nada evolucionable ni mantenible. Pan para hoy y hambre para mañana… En ese sentido la entrega inicial seguramente fue un éxito (se entregó a tiempo) pero posteriormente nos damos cuenta de la podredumbre subyacente.

Efectivamente, los condicionantes psicológicos que rodean el acto de programar importan y mucho. Un desarrollador no implementa código limpio con un látigo detrás; un profesional de la programación no ejecuta una solución bella y elegante cuando sabe que no va a ser valorada por el equipo, de quien además puede sospechar que se la puedan apropiar. No se puede crear nada, ni software, ni arte ni siquiera plantar un huerto buscando la excelencia en un ambiente desmotivador y nada creativo. En software, «bello y elegante» significa fácil y barato de evolucionar y mantener.

Esta situación se repite en demasiadas ocasiones y se seguirá repitiendo porque no se entiende lo suficientemente bien la naturaleza «mantenible» y esencialmente simple que debe tener cualquier solución software que se entrega. ¡Es como si nos vendieran un coche que no se puede reparar! Tampoco se entiende que es difícil valorar y verificar la calidad del trabajo de un desarrollador, aunque se empleen métricas por todos conocidas. Para evaluar esta calidad, hace falta mucha, muchísima experiencia en software y, lamentablemente, la dinámica empresarial en la que nos movemos hace que los managers, coordinadores y jefes ni tengan esa experiencia y estén más

centrados en tareas de gestión, eternas reuniones y en cumplir fechas «sea como sea», me temo.

Cuando la herencia envenenada explota, el equipo original ya ha sido descompuesto en pedazos (algunos incluso habrán cambiado de compañía) y el problema, que para entonces se habrá convertido en un gran problema, le caerá a otro equipo que nada tiene que ver con el asunto, al que le llega, otra vez, con tiempos cortos e insuficientes, y vuelta a repetir la misma historia, el chocar eternamente con la misma piedra; mientras, los costes siguen subiendo.

Detrás de esta dinámica hay gente que aun queriendo hacer bien su trabajo no se le ofrecen las condiciones para desarrollarlo con pulcritud y calidad. ¿Provee la compañía de las condiciones adecuadas para desarrollar software de calidad?

Ahora bien, hay quienes deben ser los responsables de ofrecer y fomentar este clima, pero estos en general están a otras cosas (como por ejemplo apretar un poquito más con algo que no estaba previsto). Es un problema también de cultura empresarial y *coaching* dentro de la organización. Aunque el desarrollador deba también favorecer con su actitud un buen ambiente laboral, el buen manager debe detectar situaciones de fricción y fomentar un ambiente de cooperación y bienestar en el equipo: forma parte de su rol.

El profesional que percibe esto y que con suficiente experiencia sabe cuál es el título final de la película, incapaz de cambiar estas condiciones y dinámicas, es en definitiva un profesional infrautilizado, un recurso malgastado y frustrado: es un empleado que sufre y que llega a casa con síndrome de estrés. Este es el perfil de ingeniero de software que padece la sensación de que la pagan por la rutinaria labor de «picar código» en lugar de por un verdadero trabajo de diseño e ingeniería.

El proyecto se podrá entregar en tiempo, pero en sí mismo es un fracaso por la estela de sufrimiento personal, falta de calidad, vulnerabilidad y fragilidad de lo entregado, lo que hará que los costes de su mantenimiento y evolución a medio y largo plazo sean muchísimo mayores.

En la economía actual sólo podemos elegir entre competir con calidad o cantidad. Yo prefiero lo primero (y confío en que las empresas serias también), aunque al mismo tiempo sé que la cantidad depende también de

una cuestión de productividad.

He aquí una gema de sabiduría para el profesional: si detectas que estás en una situación similar, entonces ya sabes el final; tu responsabilidad es alertar a los responsables, convencer al resto del equipo de la inutilidad de trabajar de esa manera y, por qué no, un profesional no malgasta su tiempo en proyectos con los pies de barro. Si el perfil del equipo es incompatible, el manager deberá reestructurarlo, así de sencillo. Sobre el proyecto internacional que comentaba al principio: solicité mi cambio de ubicación en la empresa.

Puntos clave

- El desarrollo de software es una actividad altamente creativa; por tanto, necesita de un ambiente que fomente esta creatividad, no que la destruya o asfixie.
- Si hay que correr de manera extraordinaria para «entregar» como sea según las fechas previstas, es evidente el síntoma de que alguien planificó mal el trabajo y las fechas.
- El trabajar rápido y con estrés de manera crónica en el proyecto hará que la calidad del software generado sea infinitamente menor que el trabajar a un ritmo adecuado y más relajado. Esta falta de calidad se traduce en dinero: mayor coste futuro para corregir errores, mayor número de estos, mayores problemas para evolucionar el sistema, clientes insatisfechos por la falta de calidad, etc.
- Como profesionales debemos fomentar un ambiente correcto y adecuado en el equipo de trabajo; cuando es imposible tenemos que tener claro que nos perjudica como profesionales.
- El currículum sólo lo da la calidad, no la cantidad de proyectos mal acabados y con problemas.

Todo es cuestión de principios

«No podemos desarrollar software si no aplicamos explícita e implícitamente principios de diseño y practicamos activamente buenos hábitos de programación para que el resultado sea limpio y lo más sencillo posible. Esta es una verdad que debemos incluir en nuestro ADN profesional: la deuda técnica de no aplicarlos es tan alta que no podemos asumir desarrollar sin la aplicación de estos principios y buenas prácticas. Un código en donde estos no se pueden reconocer es un código de principiantes, no de profesionales.»

En una ocasión cayó en mis manos una aplicación cuyos autores contaban con muy buena consideración en la compañía. El software funcionaba, eso era evidente ya que estaba en explotación en diversas instalaciones, pero no lo era tanto la cantidad de tiempo que una compañera y yo pasamos intentando descifrar el funcionamiento del mismo para su evolución; nos habían pedido incluir nueva funcionalidad y a pesar de ser un sistema no muy complejo, nos costó muchísimo esfuerzo entender la estructura general y cómo y dónde poder aplicar los cambios.

Cuando lees buena literatura puedes apreciar e intuir mucha más información de la que el autor plasma literalmente en el texto; un cuento, sin ir más lejos, es mucho más que la historia de unos personajes carismáticos con final feliz: hay una moraleja y lección que se extrae de él pero que está implícita y hay que descubrirla. La buena literatura es así, la buena narrativa dice mucho más que aquello que se lee explícitamente.

En software pasa exactamente lo mismo, por esa razón considero que los desarrolladores tenemos algo de «autores»: cuando leemos código realizado

por otros podemos entrar en la mente enmarañada y confusa de los programadores o bien introducirnos en un paraíso de calma y simplicidad de quien se ha preocupado en dejar las cosas limpias y fáciles de digerir. El buen código, al igual que la moraleja del cuento, revela implícitamente la «intención de diseño».

Esta es una de las grandes diferencias entre el mal y el buen software. O lo que es lo mismo, una aplicación que realiza operaciones complejas no tiene por qué haber sido implementada de forma complicada y hermética, todo lo contrario, hasta lo más difícil se puede y debe solucionar de manera simple. Es cierto que el funcionamiento puede ser igual de correcto para una aplicación resuelta de manera compleja y otra igual que hace exactamente lo mismo pero cuyo diseño es más simple, elegante y sencillo. Los sistemas pueden funcionar en ambos casos, de eso no hay duda, pero el esfuerzo, dedicación y seguramente frustración para entender lo desarrollado pueden ser muy diferentes. Todavía no termino de comprender por qué algunos desarrolladores que pecan un poco de soberbia y presumen de lo buenos y gurús que son, terminan escribiendo código difícil de entender para todo el mundo…, menos por ellos mismos. No sé si existe relación o no, me gustaría pensar que no; sin embargo, el resolver un problema complejo de manera sencilla tiene más de genialidad que resolverlo intrincadamente con algo difícil de entender. La genialidad de un buen desarrollador de software está en saber encontrar soluciones sencillas. Esto para mí es un principio incuestionable.

En ocasiones nos encontramos con código lleno de comentarios, ficheros readme.txt como base de la documentación, extensos diagramas que son imposibles de entender, nombres de clases, de funciones y variables totalmente crípticas: todo eso supone un tremendo laberinto sin salida que impide a cualquier software su evolución y comprensión por otros. La buena noticia es que esto se puede evitar y lo sorprendente es que las técnicas y hábitos para evitar software hermético y enmarañado son sencillos de aplicar.

Un error fundamental de cualquier autor de software es la creencia de que nos acordaremos de cómo hemos hecho las cosas semanas y meses después: nada más lejos de la realidad. En mi caso, dudo que en un mes pueda retomar rápidamente el código desarrollado durante un ciclo intenso de trabajo si este

no cuenta con la suficiente coherencia y claridad. Si no nos preocupamos por esto activamente iremos dejando sin darnos cuenta enormes piedras por el camino con las que tropezaremos pasado mañana o le complicaremos la vida al compañero que retomará nuestro trabajo. Este código «sucio» costará más dinero desarrollar y mantener, restándole ventajas ante un software competidor. Por poner un ejemplo, la presencia excesiva de comentarios en el código, precisamente, revela que algo no se está resolviendo de manera sencilla y autodescriptiva.

Como autores podemos escribir software de manera enmarañada o bien clara y elegante que hasta un niño pueda entender la moraleja implícita… Por eso precisamente somos autores además de desarrolladores profesionales: nuestra misión es hacer que el texto (código) que escribimos sea entendible por nosotros mismos o por otros miembros del equipo pasado un tiempo.

Esto requiere de un esfuerzo extra en nuestro trabajo diario, y es que debemos pensar como «autores» además de como desarrolladores. Escribimos para que nos lean (y la mayoría de las veces el lector será uno mismo cuando más adelante nos requieran cambios en el proyecto).

Al igual que un escritor cuenta con técnicas y recursos narrativos con los que penetrar mejor en la mente del lector, los programadores contamos con herramientas y estrategias para hacer que nuestro software sea más entendible, manejable y, en definitiva, que se pueda mantener fácilmente.

No podemos desarrollar software si no aplicamos explícita e implícitamente principios de diseño y practicamos activamente buenos hábitos de programación para que el resultado sea limpio y lo más sencillo posible. Esta es una verdad que debemos incluir en nuestro ADN profesional: la deuda técnica de no aplicarlos es tan alta que no podemos asumir desarrollar sin la aplicación de estos principios y buenas prácticas. Un código en donde estos no se pueden reconocer es un código de principiantes, no de profesionales.

Dos de los principios de diseño que más uso y que entiendo que son más útiles son el de inyección de dependencias (*dependency inyection principle* o DIP) y el de factorías de clases. Sólo aplicando estos dos conseguimos aumentar en varios niveles la calidad de nuestras aplicaciones por el grado de desacoplamiento entre módulos que conseguimos y la facilidad con que

podemos crear pruebas sobre las distintas partes del sistema; esto es sólo la punta del iceberg…

Contrariamente a lo que algunos piensan, estas buenas prácticas de diseño no persiguen hacer un software académico, ni mucho menos; todos hemos recibido cierta formación en nuestra etapa de estudiantes sobre patrones, pero pocos los tenemos realmente en cuenta cuando afrontamos la solución de un problema.

Dentro del mundo de desarrollo ágil, los principios S.O.L.I.D se consideran muy buenas prácticas en las que basar nuestras soluciones, ¿la razón?, porque de su buena aplicación depende que seamos capaces de generar un código mantenible y testeable. La aplicación correcta de S.O.L.I.D es un arte en sí mismo, para lo que recomiendo hacer pequeños programas de ejemplo donde se aplican cada uno de ellos hasta que los entendamos completamente o bien leer mucho software desarrollado por manos más expertas.

Este no es un libro técnico (o al menos no excesivamente), pero lo que me limito a indicar aquí es que existen técnicas que nos permiten crear software limpio y evolucionable y que, además, son fáciles de aplicar; es más, considero más importante para nuestro trabajo el poder desarrollar código limpio que dominar al 100% un lenguaje o tecnología.

Al igual que cuando uno aprende una nueva forma de hacer algo, corrigiendo seguramente vicios anteriores, la aplicación de estos principios de diseño nos obligará a modificar nuestra manera de «pensar» a la hora de solucionar los problemas habituales: debemos hacer un mayor esfuerzo de abstracción; esto sólo ocurre al principio, en el momento en que algo es novedad; al poco tiempo, lo habremos incorporado como hábito y el buen hacer saldrá «por sí solo». Me parece importante insistir en esto: si hay que hacer un pequeño esfuerzo al intentar aprender e incorporar estos nuevos hábitos, éste se hará sólo al principio, después lo rentabilizaremos por el mejor trabajo realizado.

Para que os hagáis una idea, la aplicación que heredé y que comentaba al comienzo del capítulo, aún funcionando estupendamente, adolecía de los siguientes problemas:

- Eran habituales las clases extraordinariamente grandes e incluso de varios cientos de líneas de código (esto demuestra poca separación funcional).
- Se hacía un uso intensivo de nombres crípticos de métodos y variables, tipo _pta, _idc, cosas así (falta de legibilidad: lo que complica mucho leer y entender cualquier código).
- Se intuía que había funciones que aparentemente hacían exactamente lo mismo (se duplica funcionalidad: más código innecesario y ausencia de la suficiente abstracción).
- El acoplamiento entre los distintos módulos del sistema era exageradamente alto (esto impedía probar algo independientemente y hacía imposible modificar nada sin daños colaterales).
- No pude identificar ningún tipo de patrón de diseño (lo que evidencia ausencia de buenos hábitos).
- Leía clases y clases pero no terminaba de entender la relación entre ellas (falta de claridad en su definición).
- La interfaz de usuario estaba increíblemente acoplada a todos los módulos de la aplicación (lo que provocaba que para probar algo hubiera que arrancar la aplicación completamente y hacer varios clics hasta llegar al punto exacto de depuración).
- No había ningún mecanismo de detección de errores: si fallaba el acceso a la base de datos, la aplicación sufría un *crash* catastrófico (incapacidad de trazar los problemas).
- Y, por supuesto, no había ni una sola prueba automatizada…

¿Nos damos cuenta de que este tipo de problemas son ajenos a la tecnología particular que se haya usado? Los mismos errores se pueden cometer en C#, PHP, Java, etc.

Aún funcionando bien, es evidente que la solución podía haberse realizado con muchísima mejor calidad. El tiempo que no se dedicó a mejorar el diseño y a crear pruebas automatizadas que respaldaran la aplicación seguramente era visto en esos momentos como un «coste»; no obstante, este coste extra vuelve al proyecto con efecto boomerang cuando el cliente

solicita una revisión un año después y esta revisión cuesta el doble, el triple o más que si inicialmente se hubiera enfocado el proyecto de manera más profesional. De nuevo, pan para hoy y hambre para mañana…

Este es el típico escenario en que una aplicación sólo puede ser mantenida y evolucionada por el mismo desarrollador que la había realizado. Puede parecer sorprendente lo que voy a decir, pero hay quienes creen que hacer algo complicado «intencionadamente» les hará imprescindibles: tremendo error, nada más lejos de la realidad. De hecho, en *El Libro Negro del Programador* hay un capítulo dedicado exclusivamente a este tema relativamente frecuente y perjudicial para quien cree eso.

Muy a mi pesar pasé varias semanas intentando crear el entorno en el que se esperaba que la aplicación funcionase hasta que descubrí que se había dejado escrito fijo en el código (lo que llamamos *hard-coded*), en un lugar recóndito del mismo, una dirección IP donde supuestamente debería estar un servidor jBoss «escuchando» con los *java beans* correspondientes desplegados. Más tarde localicé varias contraseñas escritas también de manera fija. ¿Os imagináis la incertidumbre y desesperación al tener que pasar horas y horas buscando este tipo de cosas entre cientos de líneas de código? ¿Es eso trabajo productivo? Sencillamente se podría haber evitado.

A pesar de todo esto y paradójicamente, los clientes que habían adquirido el sistema estaban relativamente satisfechos con su funcionamiento y con razón, ¿o es que miramos cómo está construida la mecánica de un coche cuando decidimos comprar uno? Y no hablo de clientes cualesquiera: del buen funcionamiento del sistema dependía que la compañía cliente ganara o perdiera bastante dinero.

El resultado de este tipo de software de juguete, nada profesional, es que apenas pudo incluirse la nueva funcionalidad sobre la que se tenían tantas expectativas, se gastó muchísimo tiempo (dinero) en meter con calzador lo poco que se pudo hacer o bien yo no fui lo suficientemente sagaz para hacerme con la aplicación en poco tiempo…

Los principios, patrones de diseño y buenas prácticas no son una moda pasajera o recursos complejos y de élite que nos complican la vida a los desarrolladores, sino que son la tabla de salvación que nos permitirán escribir código elegante que nosotros más adelante o cualquiera que herede el sistema

o parte del mismo va a entender con mucha mayor facilidad. De aplicarlos bien, no aplicarlos o también usarlos exageradamente mal depende la rentabilidad que podamos obtener por el software realizado.

Me sorprende la cantidad de tiempo que dedicamos a aprender novedosas tecnologías en forma de nuevos entornos, nuevos lenguajes, etc. y lo poco que dedicamos a pulir nuestra maestría en la aplicación y entendimiento de los principios y patrones que harán de nuestro software un sólido edificio. Apenas conocemos algo, pasamos a conocer otra cosa, y otra y otra, sin querer alcanzar una verdadera maestría y experiencia en algo concreto. Quizá eso explique por qué es difícil de encontrar proyectos verdaderamente bien estructurados y planteados. Quiero decir, ¿conocéis a alguien que lleve más de cinco años programando con la misma tecnología? Y si es así, lo habitual es que tendamos a hacer siempre las mismas cosas del mismo modo que conocemos, por pura inercia.

Tengo que reconocer que esta experiencia que resumo aquí me cogió en una época en la yo no era capaz de apreciar este tipo de cosas; aun conociendo ciertas buenas prácticas, en ese momento ni siguiera enjuicié la aplicación como mala: sencillamente el problema lo tenía yo porque no pude hacerla funcionar en el tiempo previsto. Con el tiempo y algo más de experiencia comencé a intuir que el problema no estaba tanto en mí sino en el trabajo de los autores que realizaron la aplicación que parecía que se habían obstinado en hacerla demasiado enrevesada e ininteligible (tampoco se trataba de enviar un cohete a la luna, digo yo); ahora mismo cuando pienso en este asunto mi conclusión es que aquel software no era para nada profesional a pesar de haberse realizado en el contexto de una gran empresa.

El buen desarrollador debe ser capaz de generar soluciones elegantes siempre a partir de la aplicación de principios y patrones bien identificados además de buenas prácticas de arquitectura. Esta «intención de diseño» se debe poder leer bien tanto por nosotros mismos como por cualquiera que herede el código. Igual que para aprender a sumar primero debemos conocer los números, sólo podemos programar correcta y profesionalmente si nuestras herramientas son principios de diseño que aplicamos coherentemente en la aplicación… y es que todo es cuestión de principios.

Por último, existe el mito muy extendido según el cual un software cuanto

más complejo más sofisticado; seguramente en muchos casos no se ha sabido simplificar lo suficiente. En mi opinión, la genialidad consiste en saber encontrar una solución sencilla y abordable a algo realmente complejo.

Puntos clave

- La genialidad de una pieza de código que resuelve algo está en que se ha sabido encontrar una solución lo más sencilla posible.
- El tener presente un buen hábito de refactorizar nos permite hallar soluciones cada vez más sencillas y elegantes.
- No se trata de buscar «porque sí» una solución sencilla: es que esta lleva aparejada una mejor legibilidad y permitirá un mejor mantenimiento y evolución del proyecto, ahorrando costes y haciéndolo rentable.
- Si queremos seguir manteniendo amigos, nada mejor que permitir que otros hereden un proyecto software bien hecho y entendible; de lo contrario, puede que surja quienes nos odien a muerte…
- En ocasiones existe un exceso de comentarios (que no de documentación), lo cual revela que el programador (autor) tiene necesidad de explicar algo que no es suficientemente evidente en el código. Algunas veces son necesarios, pero no por norma.

Una vuelta de tuerca a la ley del cambio

«Podemos afirmar casi sin margen de error que cuanto más popular sea una librería más evolucionará con el tiempo, lo cual es bueno, pero muy malo si nuestro software que se basa en ella no lo hace a igual ritmo. Se crea así un grado de dependencia que en algunos casos incluso puede poner en peligro el funcionamiento mismo de un sistema.»

En *El Libro Negro del Programador* se habla e insiste extensamente sobre la naturaleza de La Ley del Cambio que afecta de manera muy especial a nuestra actividad: la probabilidad de que nuestro software deba evolucionar y cambiar es mayor cuanto más tiempo de vida tenga. Esta es una ley imperturbable y es raro el que no la haya podido comprobar y sufrir en sus propias carnes.

Como desarrollador que intenta hacer su trabajo lo mejor posible, estoy convencido de que la mayoría de los problemas con los que nos tropezamos una y otra vez provienen del hecho de que no hemos digerido aún la naturaleza mutable y cambiante de cualquier software que implementamos.

Hay muchas razones por las que una solución en explotación tiene y debería cambiar, entre ellas la aparición de bugs no detectados previamente, la exigencia de nuevas funcionalidades, la adaptación a nuevas normativas legales, la misma evolución del mercado, si es un producto internacional la idiosincrasia local hará que se tengan que hacer adaptaciones específicas y, por qué no, cuanto más éxito tenga un producto más cambios se les exigirá. ¿Nos imaginamos un Facebook exactamente igual que el de hace años? En ocasiones este nivel de cambios y adaptaciones se produce de manera

vertiginosa; en el mundo Android se habla de la «fragmentación» de su API, la cual es bastante alta al aparecer cada pocos meses una nueva versión. Esto no es malo en sí mismo, ya que es de suponer que cada nueva release implementa más y mejor funcionalidad y resuelve problemas de la anterior. Lo que hay que destacar aquí es que el «cambio es continuo».

Precisamente el concepto de *continuous delivery*[1] se basa fundamentalmente en esto: salir con algo, mejorarlo y evolucionarlo continuamente, pero para ello el software debe poder ser mejorado y evolucionado con relativa facilidad.

Aunque la inercia empresarial dominante es la de trabajar una vez y rentabilizar al máximo el producto generado (lo cual no digo que esté mal ni mucho menos), en software este esquema no funciona necesariamente así: cuando se lanza al mercado un producto, cuanto más éxito tenga más cambios y mejoras se le va a exigir. De este modo, la necesidad de aplicar cambios puede ser un síntoma de éxito. Si esto es así, la capacidad de nuestro software de poder ser mejorado garantizará el éxito futuro.

No hay que ser muy avispado para darse cuenta de que la necesidad de poder modificar y evolucionar nuestro software debe ser la piedra angular de nuestros desarrollos: según el manifiesto ágil, el cambio siempre es bienvenido[2], yo diría que hasta imprescindible para el éxito en una economía global en la que sobrevive el que mayor capacidad de adaptación (cambio) presenta.

No obstante, como desarrolladores no nos damos cuenta de que esta dinámica de adaptabilidad al cambio la presentan también todas aquellas librerías, componentes de terceros y módulos externos que usamos y sobre los que construimos nuestras aplicaciones; este hecho tiene profundas consecuencias en el desarrollo y vida de nuestras aplicaciones.

Al igual que el motor de un coche no serviría de nada si no viniera acompañado del resto de componentes de un vehículo, cuando desarrollamos una nueva aplicación la hacemos sobre el trabajo ya realizado por muchos otros, esto es, el software es en cierta medida «de naturaleza piramidal», usamos frameworks, librerías, módulos, APIs basadas en servicios webs, servicios online y un larguísimo etcétera hasta llegar a la misma construcción externa que constituye el sistema operativo donde se ejecuta nuestro

software.

En un sistema en el que estoy trabajando en el momento de escribir esto, he contabilizado multitud de librerías externas (logging, planificador de tareas, inyección de dependencias, compresión de ficheros en formato zip, librería para generar ficheros excel, framework de componentes web, una librería muy popular de javascript y seguro que se me escapan algunas). De este modo, conseguimos la funcionalidad que debemos implementar ensamblando y usando componentes funcionales desarrollados por gente a la que nunca conoceremos personalmente pero sobre los que esperamos tener la garantía de su buen funcionamiento y estabilidad.

En este sentido la capacidad de desarrollar software es también la de saber ensamblar correctamente módulos heterogéneos.

Salvo que cometamos uno de los errores del principiante que consiste en creer que es más rápido hacerlo uno mismo que el esfuerzo de aprender a usar un componente ya desarrollado, esta naturaleza ecléctica es la común de cualquier software profesional y es precisamente uno de los puntos fundamentales por los que las tecnologías de la información evolucionan tan rápidamente: nos gusta compartir y además en la evolución y adaptación a los nuevos mercados y necesidades está la clave de nuestro negocio; nuestro trabajo es muy interdependiente del trabajo de otros desarrolladores.

Es más, me atrevería a afirmar que es mayor el esfuerzo que dedicamos a aprender a usar toda esa constelación de componentes, frameworks y librerías que orbitan alrededor de nuestra aplicación que el esfuerzo que necesitamos para aprender un lenguaje de programación o una tecnología particular.

Es aquí donde se presenta uno de los mayores problemas (y errores) que tenemos los que desarrollamos aplicaciones: si es cierto que nuestro software debería y tiene que evolucionar y cambiar, esa constelación de componentes que usamos lógicamente cambiará también, la «ley del cambio» también les aplica a ellos puesto que no dejan de ser artefactos software.

Parece evidente aunque no lo es tanto la naturaleza de la relación que asocia nuestro software con el de los módulos externos que decidimos usar: en muchas ocasiones nos vemos lastrados precisamente por la misma evolución del software de terceros que hemos elegido sin hacer demasiadas consideraciones.

¿Debemos utilizar la última actualización de una librería externa cuando se le han detectado ciertos errores y vulnerabilidades? Lógicamente sí, pero en ocasiones esto representa un enorme coste ya que hay que considerar el actualizar la librería y al mismo tiempo asegurarnos de que todo el sistema sigue funcionando correctamente. En una ocasión vi cómo se actualizaba demasiado a la ligera un componente externo que comenzó a dar muchos problemas en producción: sencillamente no se había leído con atención que cierta funcionalidad quedaba descartada y obsoleta en la nueva versión…

Es uno de los grandes errores que cometemos cuando decidimos qué librerías usar en nuestra aplicación. Se da la circunstancia de que no sabemos evaluar bien la idoneidad de un componente u otro y en ocasiones nos dejamos guiar por los gustos particulares de ese momento o su actual popularidad.

Otras veces es el impulso de usar cosas nuevas «porque sí» el que nos lleva a tomar decisiones equivocadas con resultados a medio plazo catastróficos.

Es de sentido común darse cuenta de que si nuestro software tiene que estar preparado y listo para ser modificado, las librerías y componentes externos de terceros que usamos también. ¿Tenemos esto en cuenta cuando los seleccionamos? ¿Tenemos en consideración el coste de adaptación a las nuevas releases de esos componentes a medio plazo?

Podemos afirmar casi sin margen de error que cuanto más popular sea una librería más evolucionará con el tiempo, lo cual es bueno, pero muy malo si nuestro software que se basa en ella no lo hace a igual ritmo. Se crea así un grado de dependencia que en algunos casos incluso puede poner en peligro el funcionamiento y la correcta ejecución de un proyecto.

Un desarrollador profesional debe centrar sus esfuerzos en implementar código funcional, en poner en marcha funcionalidades de valor para el cliente. Para ello debe usar en la medida de lo posible librerías de terceros bien conocidas, documentadas y estables, pero siempre realizando un exhaustivo análisis de la idoneidad de su uso en el contexto del sistema que está en desarrollo.

Se pueden cometer errores catastróficos con una mala elección cuando nos llevamos la sorpresa de que un componente externo que usamos ha

dejado de ser mantenido y soportado, lo cual puede ocurrir de igual modo si pertenecía a una compañía grande y consolidada o bien era mantenido por un grupo de desarrolladores de modo altruista y para la comunidad. El riesgo es el mismo. Lo importante aquí y que quiero significar es que hay que tener en cuenta ese riesgo.

Esta situación me pasó cuando en un sitio web basado en Drupal decidí usar un módulo muy interesante y que funcionaba aceptablemente bien pero que fue abandonado al poco tiempo; cuando meses más tarde fui a actualizar el site, había incompatibilidades entre este módulo y otras versiones más recientes de otros, lo que me obligó a hacer bastantes cambios en la web: mucho tiempo dedicado a reestructurar y eliminar el módulo obsoleto en lugar de dedicarlo a aportar valor a la solución.

No es más que un ejemplo real (y que conste que adoro Drupal) aunque pasa en multitud de contextos distintos. Podemos ver aquí los dos contrapesos de la balanza: módulos que evolucionan (y mejoran) rápido, lo cual es bueno, indica dinamismo, pero al mismo tiempo provoca traumatismos cuando alguno es abandonado y presenta conflictos con otros.

Antes de elegir una librería de terceros, debemos hacernos algunas preguntas fundamentales si no queremos plantar la semilla de futuros problemas:

- ¿Cuenta con una comunidad suficientemente amplia?
- ¿Es una librería relativamente nueva y, por tanto, previsible de una rápida evolución?
- ¿La puedo sustituir fácilmente por otra equivalente?
- ¿Existe suficiente documentación, foros y comunidad que me permita usarla con facilidad?
- ¿Han respetado las nuevas releases la compatibilidad hacia atrás o son demasiado disruptivas?

No podemos avanzar en un software profesional si cuando elegimos los componentes sobre los que vamos a construir nuestra aplicación no los analizamos de arriba abajo en estos términos: el coste a medio plazo de una mala elección puede ser muy alto.

Es una constante en software que ciertas tecnologías, librerías, APIs públicas y componentes se terminen convirtiendo en obsoletas, lo cual puede ser en sí mismo un síntoma de que han aparecido más y mejores elementos que los sustituyen, pero es todo un problema cuando lastran nuestras aplicaciones al poco tiempo de lanzarlas. Necesariamente tenemos que considerar este riesgo.

El caso más grave que conozco es el de un sistema, digamos, muy grande, en el que se decidió muy a la ligera usar una tecnología de Microsoft en ese momento aún muy preliminar y recién salida de la cocina. Al año se lanzó una nueva versión de esa tecnología y los costes de adaptación fueron enormes (no había compatibilidad hacia atrás). No solo eso sino que llegó la ola *cloud* y por fuerza el sistema tenía que ser desplegado en la nube; la frustración fue total y los problemas tremendos cuando se descubrió que en ese momento esa tecnología en particular era incompatible con la solución de *cloud computing* de Microsoft. De nuevo esto es otro ejemplo doloroso pero real de lo que puede pasar cuando se toman ciertas decisiones y elecciones sin profundizar en las consecuencias a medio o largo plazo. Lo lamentable es que los criterios para elegir esa tecnología fueron más emocionales que técnicos.

El error fundamental que cometen muchos desarrolladores noveles es usar librerías o tecnologías con la única razón de que les gusta, nada más. El desarrollador profesional las elige teniendo en cuenta todas las consideraciones anteriores y previendo en la medida de lo posible el tiempo de vida y evolución de la aplicación en desarrollo.

No usamos una librería X porque estamos encantados con ella o porque ahora está muy en boga y es lo más *cool*, sino porque encaje bien en la naturaleza del proyecto que estamos desarrollando.

Me temo que hay veces que nos dejamos llevar demasiado por modas pasajeras sin evaluar en detalle las consecuencias. Si somos profesionales, debemos estar al tanto de las tendencias pero debemos justificar completamente su idoneidad o no en el proyecto en el que trabajamos actualmente.

En este sentido podemos decir que es aplicable al software el concepto de «obsolescencia programada»[3]: queramos o no, el tiempo de vida de la

mayoría de las aplicaciones es limitado. Podemos (y debemos) esforzarnos mucho por ser buenos *clean coders* en todo lo que esto representa, diseño, codificación, aplicación de principios, buenas prácticas y hábitos, el resultado será que alargamos notablemente el tiempo de vida y explotación del sistema aunque este nunca será para siempre ya que el mismo mercado y economía habrán dado la vuelta en sólo una década.

Es de ingenuos pensar que las librerías y componentes en las que se basa nuestro software van a permanecer intactos en los próximos dos años (me atrevería a decir que incluso en los próximos meses); quizá algunos de ellos en cinco años hayan desaparecido o serán abandonados (es el efecto que yo llamo *co-maintainers-wanted*). Me temo que hay pocas excepciones a esto.

Cuanto más reciente e incipiente sea la tecnología que usamos más aumenta la probabilidad de que este escenario se produzca. Este axioma de nuestra profesión no nos gusta en absoluto y algunos podrán decir que están en contra de que sea así, aunque a veces confundimos deseos con verdades.

Lógicamente tampoco vamos a volver al mundo del Cobol o al ANSI C, pero siempre en el término medio está la virtud: en cualquier elección de tecnologías o terceras librerías y componentes debemos evaluar a medio y largo plazo este tipo de consecuencias.

Aun eligiendo correctamente los componentes a usar según la naturaleza del proyecto debemos tomar siempre en cuenta ciertas medidas anticipatorias y preventivas. En lo posible tenemos que aislar la funcionalidad que nos da cierto componente pensando siempre en la posibilidad de su sustitución.

En software apenas está implementado el concepto de estándares industriales según el cual los fabricantes construyen vehículos (nuestro producto) a los que se les puede acoplar ruedas de diversos proveedores (las librerías); ¿pero no está para eso precisamente la inyección de dependencias o DIP[4] (*dependency injection principle*)?

Puesto que las librerías externas que usamos pueden cambiar a un ritmo mucho mayor del que evoluciona nuestra aplicación, pueden desaparecer, dejar de ser mantenidas, etc. debemos abstraer al máximo su uso en nuestro sistema de manera que este sólo consuma su funcionalidad mediante interfaces genéricas sin hablar directamente con el componente. Tenemos que desacoplar al máximo nuestro software del resto de componentes de terceros.

Esto nos permitirá sustituirlos en un futuro de una manera menos traumática.

Puntos clave

- Cuanto mayor tiempo de vida se espere para nuestra aplicación, mayor probabilidad de que deba evolucionar y ser modificada (Ley del cambio).
- Esta ley es aún más intensa para componentes y librerías externas, ya que al otorgar funcionalidad más específica son más versátiles para evolucionar.
- Cuanto más popular y usada es una librería, más susceptible será de cambiar y generar un mayor número de versiones.
- La elección de un componente u otro no se debe hacer guiado por modas pasajeras o porque esté en boca de todos: hay que hacer una elección objetiva adaptada a las necesidades reales y futuras de nuestra aplicación aunque esto suponga usar componentes «antiguos».
- Estos terceros componentes que usamos como soporte de nuestras aplicaciones pueden morir, quedarse obsoletos o dejar de ser mantenidos.
- Debemos aislar al máximo la funcionalidad de estas librerías en nuestro sistema abstrayendo su funcionalidad mediante interfaces. Esto nos permitirá poder sustituirlas más cómodamente.

Atreverse a eliminar lo implementado

«El desarrollo de software es una tarea incremental en la que en cada paso vamos añadiendo funcionalidad pero también vamos adaptando lo anterior para acomodarlo a los nuevos requerimientos. Para ello creamos nuevas secciones de código y modificamos el existente. El objetivo no es sólo crear algo que funcione, también encontrar la solución más simple con el mejor diseño para los requerimientos que nos han dado. Sólo daremos con esa solución si estamos dispuestos a modificar en mayor o menor grado lo que ya hemos implementado. Modificar significa en ocasiones tirar a la basura y eliminar piezas de código.»

Podemos imaginar un pintor al que le encargan un cuadro con breves y vagas descripciones: el cliente no tiene claro exactamente qué es lo que desea. En un primer acercamiento, el autor crea un esbozo, un simple borrador y tras un tiempo se lo muestra al cliente. Este rectifica, da nuevas instrucciones y afina en algunos detalles en los que nunca había pensado. En esta segunda vuelta el autor ya tiene una mejor idea de lo que debe dibujar. Se pone manos a la obra y tomando un nuevo lienzo realiza un segundo acercamiento a la idea que tiene en mente el cliente. Tras enseñarle el resultado, éste vuelve a indicar nuevas instrucciones: cada vez está más cerca de llegar al cuadro perfecto que desea. En esta tercera ocasión el autor completa en un tercer lienzo el cuadro que realmente se le ha pedido: nada que ver con aquel lejano borrador que esbozó al principio.

Desarrollar software tiene una dinámica muy parecida al ejemplo que describimos del pintor y su cliente con la diferencia fundamental de que

cuando escribimos código, cuando solucionamos el problema que tiene el cliente, no tenemos que tirar a la basura lienzo tras lienzo y comenzar desde cero; en la mayoría de las ocasiones conseguimos llegar a la mejor solución de manera incremental sobre lo ya implementado previamente, lo que casi siempre requiere realizar un ejercicio de destrucción.

Uno de los principios fundamentales del software es que cualquier solución, sistema, módulo, librería, etc. «aumenta en complejidad» de manera natural. Bien porque con el tiempo se han introducido nuevas características a implementar bien porque el diseño inicial no encajaba a la perfección con la evolución de la solución; en ambos casos el resultado suele ser el mismo: lo que tenía un diseño prístino con el tiempo se va deteriorando necesariamente; aquello que nos quedó perfecto termina degenerando en cierta medida cuando sobre ello construimos más y más funcionalidad. Esto es lo que suele ocurrir aunque es nuestra responsabilidad evitarlo.

Esta tendencia natural a deteriorarse (*software rot*[6] en inglés) existe igualmente cuando abordamos nuestras soluciones por fases o *sprints* de trabajo. Siempre que volvemos a una sección del sistema para modificar su comportamiento o agregarle características al final lo hacemos a costa de convertirlo en algo más complejo. Con el tiempo esta complejidad va creciendo y termina generando software muy deteriorado, difícil de entender y evolucionar y en el peor de los casos prácticamente imposible de mantener. Una regla fundamental del software profesional es que tiene que permitir dedicar menos esfuerzo a mantenimiento que a incorporar nuevas y mejores características[5]; la cuestión está en cómo conseguir ese mundo ideal en el que el coste de mantener un sistema en producción sea menor que desarrollar nueva funcionalidad sobre él.

Esto, digo, es natural que ocurra y dudo que haya alguien con algunos años de experiencia que no se haya encontrado con esta situación alguna vez. Es más, se da la circunstancia de que si se evaluara el tiempo y coste de mantener algunos proyectos se llegaría a la conclusión de que ¡habría resultado más barato implementarlos de nuevo! No obstante, a ver quién es el que se atreve en un ejercicio de sinceridad decirle a su jefe que es mejor tirar a la basura lo que se ha hecho hasta el momento (aun siendo más barato que intentar evolucionarlo). No deja de ser un tanto paradójico.

Si natural es que una aplicación aumente en complejidad «si no hacemos nada», lo es también que las pruebas que la soportan aumenten en tamaño, número y, de nuevo, también en complejidad. Es interesante ver cómo no se tiene en cuenta que hay que cuidar la calidad de las pruebas, ya que ello determinará que tengamos más o menos problemas en seguir evolucionando la solución. Personalmente me he encontrado alguna vez diciéndome a mí mismo cosas como «puff, si simplifico esto así, voy a tener que cambiar todas estas pruebas». La decisión correcta es hacerlo si con ello vamos a dejar el sistema mejorado en algún aspecto.

Un programador profesional tiene asumido en su ADN que no se puede desarrollar software sin el respaldo de tests automatizados, pero ¿nos hemos dado cuenta de que también debemos hacer mantenibles, legibles y simples las mismas pruebas? De igual modo que nuestra solución se va enrevesando con el tiempo, también lo harán las pruebas que la soportan.

A lo largo de estos últimos años he podido comprobar por mí mismo la razón fundamental por la que el código de compañeros (y el mío propio) se ha ido haciendo más complejo con el tiempo y no es otra que nuestra tendencia a no querer eliminar y modificar en profundidad secciones de código que en su día dimos por cerradas y completadas: en definitiva, hay una razón «emocional» que nos impide eliminar cosas que nos costó horas de esfuerzo realizar.

¿Quién no ha sufrido alguna vez esa desazón al tener que eliminar una clase sobre la que estuvo trabajando mucho tiempo con anterioridad? ¿No nos obstinamos a veces en mantener intacto algo en el código por la razón emocional de que lo hemos parido nosotros mismos? ¿A quién no le da especial pereza mejorar tests? Seguramente sea la misma sensación del pintor cuando tiene que tirar a la basura los lienzos con su primer y segundo intentos del cuadro perfecto o la del escritor cuando se tiene que deshacer del primer borrador de su libro: en definitiva parece que el esfuerzo de hacer todas esas pruebas de concepto no sirvió para nada. Sin embargo no somos suficientemente conscientes de que «precisamente» por el esfuerzo de esos primeros intentos llegamos a una mejor solución final, sea un cuadro, un libro o un sistema software. Nada sale perfecto desde el primer momento, sino que lo vamos mejorando incrementalmente a medida que conocemos mejor la

naturaleza del problema a resolver.

Un desarrollador de software profesional no duda en eliminar lo que sea necesario siempre que vaya a ser sustituido por una idea mejor que añada simplicidad, mejor diseño o mayor legibilidad a la solución. Se puede alegar que haciendo esto se incurre en un extra de tiempo y esfuerzo, pero precisamente al simplificar y mejorar estamos multiplicando la velocidad de desarrollo más adelante: sentamos las bases para ser más productivos mañana, aunque esto es muy difícil de transmitir a nuestros gestores.

En este sentido el concepto fundamental que nos falta es que «destruimos» para construir algo que va a simplificar y mejorar la solución. Cuando eliminamos métodos duplicados de clases, sustituimos una funcionalidad por una nueva librería externa, eliminamos la implementación demasiado rebuscada de un algoritmo porque hemos encontrado un modo más sencillo de hacerlo, todo, en definitiva, lo hacemos para construir algo mejor de lo que había. Y es que no todos estamos dispuestos a modificar algo en lo que hemos trabajado muy duro en el pasado para mejorarlo por la sencilla razón de que parece que eliminamos también el esfuerzo que dedicamos en su momento.

En ocasiones es nuestro propio subconsciente el que evita tener que eliminar secciones o partes completas de una solución cuando tratamos de encontrar forzosamente una alternativa sobre lo que ya hay implementado dando lugar una solución menos limpia y óptima.

Salvo que tengamos muchos años de experiencia en un sector particular, a la mayoría nos ocurre que conocemos en profundidad un determinado dominio cuando hemos trabajado en él durante mucho tiempo. ¿Alguien podría implementar de manera perfecta un protocolo de comunicaciones la primera vez que tiene que gestionar dispositivos a ese nivel? Una interfaz gráfica basada en web pensada para algunas decenas de usuarios no va a tener nada que ver en su diseño cuando tiene que escalar a miles de usuarios concurrentes. Cualquier aplicación *data-centric* no va a prever al principio todos y cada uno de los informes que el cliente va a decir que necesita cuando se haya entregado el proyecto. ¿Puede un sistema de contabilidad prever cambios normativos? Podríamos dar cientos de ejemplos aunque todos presentan lo siguiente en común: todo software evoluciona y se vuelve más

complejo con el tiempo; para mantener la legibilidad y mantenibilidad modificamos continuamente la solución (lo que implica eliminar secciones obsoletas de código). Me temo que muchos de los desastres en proyectos software suelen producirse por no entender bien esto.

En software se da la circunstancia de que profundizamos en la soluciones que debemos entregar precisamente cuando estamos mano sobre mano implementándolas: igual que el ejemplo del pintor al comienzo del capítulo, la segunda y tercera entregas se van a acercar cada vez más a lo que realmente el cliente desea, por tanto, ¿cómo podemos pensar que todo lo implementado en la primera tiene que seguir intacto en esas etapas sucesivas?

Hay ocasiones en las que nos sinceramos completamente y llegamos a la conclusión de que tal librería o módulo no puede ser mejorado de ningún modo para abarcar los nuevos cambios solicitados: el siguiente paso es eliminarlos para rehacerlos de nuevo. Esto es un ejercicio de honestidad que redundará en la calidad de nuestro trabajo.

En inglés existe un término habitual para indicar que algo se ha renovado completamente (*revamp*) y precisamente se usa para agregar valor a una nueva versión. Esta renovación no se puede hacer sin modificar en profundidad lo anterior, lo que implica necesariamente eliminar trabajo ya realizado y que funcionaba correctamente según las expectativas anteriores.

Me encanta cuando leo en las *releases notes* de una nueva versión de algún módulo o librería, eso de «fully revamped», o sea, que lo han construido de nuevo desde cero para hacerlo mejor.

A veces nuestros propios prejuicios nos hacen enfocar el software como un trabajo de construcción en el que producimos cosas: cuanto más produzcamos, mejor, por esa razón también nos cuesta dar ese «paso atrás» cuando tenemos que eliminar algo. Este paso atrás no es más que aparente, ya que como estamos viendo es una pieza fundamental para poder luego dar tres adelante.

¿Cómo le explicamos al jefe o manager que tenemos que replantear completamente esta parte de la aplicación? Seguro que más de uno se ha encontrado en esa tesitura, lo que suele dar como resultado que arrastramos módulos que debieron ser eliminados a modificados en profundidad por no ser honestos con nuestros superiores e indicar la necesidad de sustituirlos. Lo

que nos falta aquí es contar la otra parte de la historia: ¿qué conseguimos con ese acto de destrucción?

El desarrollo de software es una tarea incremental en la que en cada paso vamos añadiendo funcionalidad pero también vamos adaptando lo anterior para acomodarlo a los nuevos requerimientos (estos nuevos requerimientos en ocasiones vienen dados por una mejor comprensión del problema a resolver). Para ello creamos nuevas secciones de código y modificamos el existente. El objetivo no es sólo crear algo que funcione, también encontrar la solución más simple con el mejor diseño para los requerimientos que nos han dado. Sólo daremos con esa solución si estamos dispuestos a modificar en mayor o menor grado lo que ya hemos implementado. Modificar significa en ocasiones tirar a la basura y eliminar piezas de código.

Personalmente no he trabajado nunca en ningún proyecto que no haya tenido que ser modificado profundamente después incluso de su entrega. Mis peores experiencias provienen precisamente de una época profesional en la que mentalmente no estaba preparado para tirar a la basura partes importantes (e inmantenibles) del sistema en el que trabajaba en ese momento. Ahora sé que lo más productivo (en tiempo y en coste) habría sido volver a escribir completamente esas partes desde cero.

Por cierto, este mismo capítulo ha sufrido esta «destrucción creativa» en varias ocasiones, por lo que no tiene nada que ver con el original y primer intento de escribir sobre estos conceptos.

Puntos clave

- Todo software va a ser modificado en mayor o menor medida a lo largo de su vida.
- Puesto que va a evolucionar, tiene que contar con un diseño e implementación lo más simple y eficiente posibles, siguiendo principios de diseño y buenas prácticas.
- Cualquier nuevo cambio va a implicar con toda seguridad modificar código existente.
- Si no se hace nada, la acumulación de una modificación tras otra generará un código complejo y cada vez más enrevesado.

- Para resolver esa complejidad creciente debemos estar dispuestos a modificar en profundidad o eliminar totalmente partes completas de nuestra solución. Hacer esto sin dudar, es un síntoma de profesionalidad.

Cuando incorporar más gente es desastre asegurado

«Si incluimos nuevos miembros en el equipo de desarrollo cuando no damos abasto para llegar a las fechas de entrega, estamos empeorando notablemente la situación y lo que haremos será adelantar aún más el fiasco.»

Érase una vez… participaba en un proyecto relativamente importante a nivel internacional, pero como suele ocurrir en muchos casos, comenzó muy limitado en recursos: éramos cuatro programadores sin ninguna experiencia en este tipo de proyectos quienes nos teníamos que encargar de todo: diseño, despliegues, base de datos, reuniones con el cliente, etc.

El tiempo fue pasando y lógicamente también fue aumentando la presión por tener cierta funcionalidad vital en fechas muy ajustadas comprometidas con el cliente. El proyecto, digo, si bien técnicamente no planteaba grandes retos, sí era importante económicamente ya que la parte residual que era el software daba cobertura al despliegue de miles de dispositivos en campo, que era la parte fundamental y económica del proyecto. Si el software no funcionaba bien, todo lo demás peligraba.

Cuando un proyecto comienza subestimado en recursos es difícil que termine bien, como bien se evidenció en aquellos meses. Las capas directivas estaban cada vez más preocupadas, insistían en conocer el alcance de los avances casi a diario con eternas reuniones improvisadas que te quitaban parte de lo que menos tenías en ese momento: ¡tiempo para trabajar! Sigo sin entender por qué se permiten las reuniones improvisadas o alguien te puede requerir de improviso en una llamada telefónica eterna. Por lo general somos

muy respetuosos con nuestro propio tiempo, pero nada con el de los demás.

La poca organización que conseguíamos tener (a menudo trabajando fines de semana) se fue diluyendo a medida que la presión aumentaba. Vivíamos en un ambiente de proyecto completamente hostil, no porque se apreciara agresividad en nadie sino porque los nervios estaban a flor de piel y estábamos muy lejos de conseguir esa relativa «paz mental» que un desarrollador necesita para conseguir un buen diseño y un buen producto: el desastre estaba casi asegurado.

El objetivo hacía tiempo que dejó de ser sentar las bases de un gran producto para la empresa sino conseguir entregar «algo» como fuese en los tiempos que otros habían comprometido sin consultar con la gente que tendría que desarrollar la solución. Es posible que el cliente exigiera esas fechas; en cualquier caso de un modo u otro lo que se había comprometido era excesivo para el tiempo dado.

Llegó entonces el inevitable momento de epifanía por parte del director de división (al que, por cierto, guardo un gran afecto). Una tarde me llamó a su despacho y los signos de preocupación eran evidentes. Dudo que imaginara siquiera la desmoralización, estrés y síndrome de *burnout* que sufríamos los que estábamos codo con codo intentado avanzar en la solución. Él también tendría sus propias presiones. Cuando se llega a ese punto no es raro que el responsable del responsable del responsable…, en fin, cuanto más alto el nivel más probabilidad real de llegar a la pregunta: ¿cuántos desarrolladores «más» necesitas?

Este es uno de los males más perniciosos de nuestra profesión en situaciones críticas, el intentar incorporar más programadores en un momento avanzado o final de un proyecto cuando la crisis y tensión son palpables. Y digo pernicioso porque cuando te hacen esa pregunta estás en un verdadero dilema: por una parte, si dices que no es buena idea entonces estarás en el punto de mira cuando el proyecto se tuerza inevitablemente más adelante (¿por qué no aceptaste más gente en el equipo?, te preguntarán), pero si aceptas incluir más programadores en ese momento no haces más que empeorar las cosas.

Si incluimos nuevos miembros en el equipo de desarrollo cuando no damos abasto para llegar a las fechas de entrega, estamos empeorando

notablemente la situación y lo que haremos será adelantar aún más el fiasco.

Incluso en situaciones normales incluir un nuevo miembro del equipo tiene su «coste» e impacto para todo el grupo, en el sentido de que se tendrá que dedicar parte del tiempo a indicarle cómo se trabaja, en qué consiste el proyecto, resolver dudas y preguntas (todo lo natural y lógico que necesitamos cuando nos enfrentamos a algo nuevo).

Una de las máximas en software es que para que la productividad sea óptima, un proyecto lo comienza y lo termina el mismo equipo. Esto es lo deseable. Si hay variabilidad en sus miembros interferiremos en mayor o menor medida en el progreso natural del proyecto. Lo ideal, digo, es que el proyecto lo comience y lo termine un mismo equipo. Nada peor que una alta rotación de personal en un proyecto.

Trabajé como responsable de mantenimiento de un sistema desplegado en un cliente de esos «importantes», para cuyo trabajo se necesitaban tres personas ya que estaba en continua evolución con nuevas peticiones. Quizá ha sido el proyecto en el que más tiempo he estado trabajando. Pude contar en una ocasión hasta doce desarrolladores distintos que habían intervenido en algún momento en el mismo; puesto que no había habido nunca un líder claro que se encargara de su organización, doce desarrolladores significaba en ese caso doce formas de hacer las cosas, por lo que algunas partes del sistema se habían convertido en un popurrí insostenible. Esto es el caso extremo de lo que ocurre cuando la rotación en un mismo proyecto es alta.

En software ocurre más que en ninguna otra profesión que aun entregando una solución que funciona, según el tiempo que se haya tenido para desarrollarla, la calidad de lo entregado puede ser muy dispar: si los tiempos han sido muy ajustados ese «ahorro» inicial de tiempo pasará factura más adelante por el mayor esfuerzo que se requerirá luego en mantenimiento. Esto es como una constante de la naturaleza, casi inevitable e insistimos mucho en ello a lo largo de *El Libro Negro del Programador*.

Intentar incorporar más desarrolladores en situaciones de crisis es lo natural cuando las capas de decisión importantes de una compañía no conocen la naturaleza del software en profundidad: volvemos a la concepción de que si un equipo de dos personas hacen un trabajo X entonces cuatro tendrán un rendimiento equivalente a 2X, por tanto el doble de

programadores tardarán la mitad de tiempo… Aquí la parte de la historia que falta es que los nuevos miembros del equipo necesitan un tiempo de adaptación y además les robarán dedicación y esfuerzo al resto del equipo precisamente en un momento ¡en el que no hay tiempo! Incluso a veces esto cuesta ser entendido por los mismos desarrolladores.

Hay ocasiones en las que un programador se ve forzado por las circunstancias a hacer un mal trabajo; esta es una de ellas. El trasfondo de todas estas situaciones suele ser el mismo: falta total de una mínima previsión y organización. Pero, ¿podemos hacer algo al respecto? Un profesional debe adelantarse a estas situaciones y advertirlas convenientemente aunque su responsable no tenga conocimiento suficiente de estas circunstancias. Si vamos viendo que el equipo no llega, antes de que el vaso se desborde, debemos advertirlo a quien corresponda.

Es cierto que en ocasiones las compañías se ven forzadas a vender con fechas imposibles por las leyes de la competencia feroz del mercado, pero al final esa ventaja competitiva se termina trasladando en forma de extrema presión a los equipos de desarrollo que no harán otra cosa que sobrevivir intentando llegar a las fechas «como sea». Se puede entender, pero también hay que entender que un desarrollador profesional no podrá realizar un trabajo igualmente profesional y de altura en este tipo de dinámicas empresariales.

Hacer software de calidad requiere de ciertas condiciones, digamos, ambientales. Se suele decir que para conseguir lo mejor de las personas hay que plantearles retos que puedan asumir, sin desbordarlos más allá de sus verdaderas capacidades y aplicar, un poco, sólo un poco, de presión. Así es como la naturaleza humana nos hace involucrarnos de lleno en los retos. No obstante, el problema del que estamos hablando se presenta cuando se exigen retos imposibles de cumplir con una presión exagerada. Lo he visto, lo he sufrido, he leído muchos casos similares y el final es siempre el mismo: desastres personales y proyectos fallidos.

Un buen desarrollador no puede hacer bien su trabajo si no le acompañan y rodean ciertas condiciones externas en su día a día; su manager y responsable es el encargado de establecer esas circunstancias, no él. Lo malo de la historia es que cuando el proyecto se tuerce, se suele apuntar al último

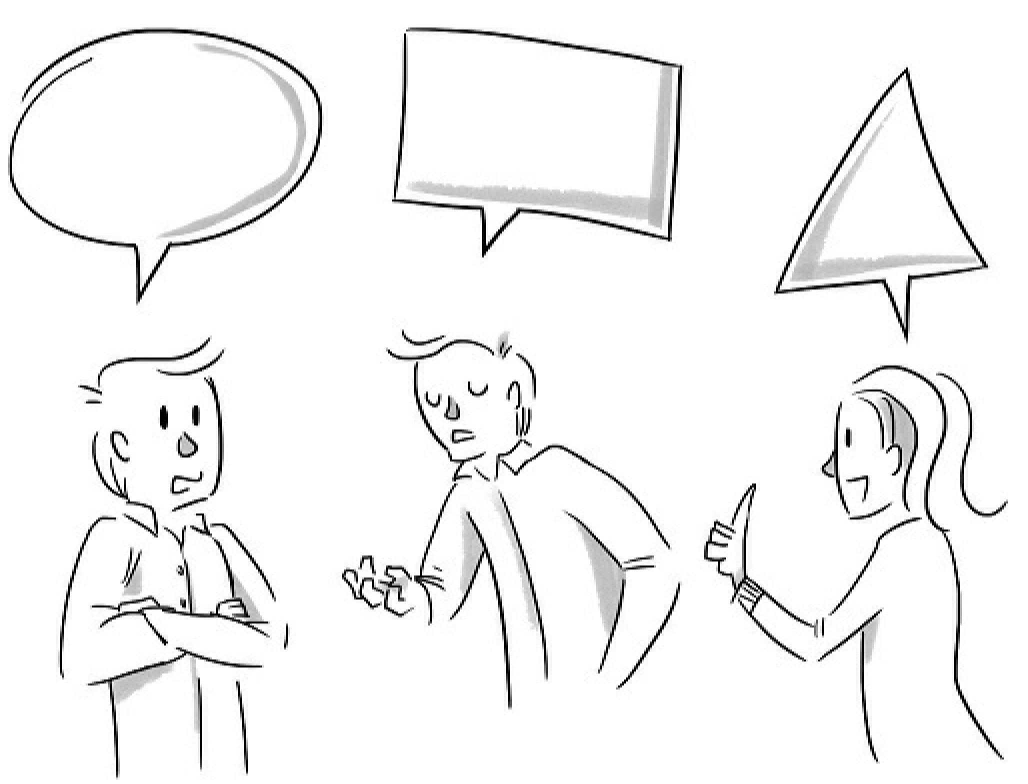
eslabón de la cadena, o sea, cuando se buscan responsabilidades se les exigen a los mismos desarrolladores por no haber avanzado a mejor ritmo. Yo siempre digo que cuando un equipo falla en un proyecto, por la razón que sea, el responsable es siempre y sin excepciones el líder o manager del equipo, así de sencillo.

Somos profesionales en la medida en que conseguimos (y nos dejan) hacer trabajos profesionales. Participando en proyectos fallidos de antemano por falta de recursos o equipos de trabajo hostiles, sólo conseguiremos quemarnos inútilmente y no avanzar profesionalmente.

Puntos clave

Cuando nos encontramos en una situación similar a la descrita en este capítulo debemos tener en cuenta lo siguiente:

- La situación empeorará cuando se incluyan nuevos desarrolladores en los momentos finales de un proyecto.
- Conviene advertir a nuestro responsable con la suficiente antelación que será difícil llegar a las fechas comprometidas (por otros).
- Resulta útil indicar que si hay posibilidad de llegar a las fechas será a costa de hacer un trabajo menos profesional.



MUCHOS PROYECTOS FALLAN POR FALTA DE ORGANIZACIÓN

Es sorprendente la cantidad de proyectos que terminan mal, que no se entregan o se tiran a la basura no por deficiencias en las competencias técnicas del equipo de trabajo, sino por una organización nefasta y una falta total de planificación.

Cuando el gestor de un proyecto es su mayor enemigo

«Gestionar significa conocer en profundidad lo que ocurre en el seno de un grupo de trabajo, ¿cómo si no vamos a tomar decisiones si no sabemos el progreso real, los problemas del día a día y las vicisitudes del grupo de profesionales que trabajan para obtener un buen resultado? Preguntar es bueno, aunque se da la circunstancia de que muchos gestores de proyectos software preguntan tanto que continuamente están incordiando y molestando al resto del equipo o bien están acostumbrados a convocar reuniones espontáneas sólo para resolver sus propias dudas; esta es la actitud de quien cree que el equipo «por debajo» de él está a su servicio, quizá por un mal entendimiento de las jerarquías, cuando en realidad es el manager quien debe estar al servicio del equipo para que este trabaje en las mejores condiciones.»

Tengo que reconocer que he sufrido experiencias malísimas con nefastos gestores de proyectos software, hasta tal punto que en su día comencé a intuir la enorme importancia de contar en un equipo de desarrollo con un rol claro y profesional que ejecute correctamente ese papel.

En cualquier grupo de trabajo cada pieza es fundamental; no es cuestión de asignar una importancia mayor a quien tiene más responsabilidad, sino darnos cuenta de que cada miembro del equipo necesita de los demás y estos a su vez le necesitan para que un proyecto común salga adelante con las aspiraciones de calidad y éxito que pretendemos. Esto es así en cualquier proyecto colectivo en el que el resultado depende de la buena colaboración de

todos sus miembros. Me temo que no todos los gestores de proyecto, coordinadores o managers lo entienden siempre así. En especial, algunos gestores de proyectos software tienen su especial idiosincrasia.

Nos guste o no, en la mayoría de compañías con equipos de desarrollo existe una persona encargada de su gestión y coordinación. No obstante, opino que muchos proyectos terminan en absolutos desastres por la sencilla razón de contar con un manager que simplemente no ha hecho bien su trabajo o bien «se le viene grande» esa responsabilidad.

En otras ocasiones, la excelencia del equipo de desarrollo es la tapadera perfecta del mal hacer del gestor; en este capítulo nos vamos a dedicar al primer caso ya que un mal responsable de equipos de desarrollo puede suponer el peor riesgo para productos y soluciones que podrían terminarse mucho mejor o con más éxito.

La regla número uno para un manager es que tiene que conocer en cierta medida la naturaleza de un desarrollo de software; no podemos coordinar la creación de algo cuyos detalles conocemos escasamente. Esto parece evidente aunque en ocasiones nos encontramos con gente que ocupa ese rol sin tener ni idea de qué se cuece y cómo es la evolución de un producto.

Quien toma decisiones relevantes sobre un proyecto tiene que conocer qué es un desarrollo software, cómo evoluciona, qué riesgos reales se adquieren con ciertas decisiones y cómo afecta la metodología elegida. Así de sencillo pero así de contundente. No es raro encontrar gestores con ninguna o nula experiencia en este sentido, en especial en grandes compañías donde existe rotación interna y la gente puede llegar de un sitio a otro «de rebote».

Un director de una fábrica de coches debe conocer la especial naturaleza de la fabricación de vehículos, el dueño de un invernadero tiene que saber cómo gestionarlo desde lo más básico; un escritor debe conocer el proceso de edición de sus propios libros y de igual modo, el responsable de un proyecto de desarrollo de software tiene que conocer en cierta medida cómo se construye una aplicación y estar familiarizado con sus conceptos fundamentales.

No tiene que ser ni siquiera experto en algunas de las tecnologías que se usen (su elección se puede delegar a alguien más experto del grupo) pero sí entender los aspectos fundamentales de un buen software.

Si no es así, ¿cómo va un desarrollador a indicar la necesidad de mejorar (*refactorizar*) ciertos módulos si el gestor de proyecto no entiende que así estamos aumentando la velocidad de trabajo más adelante? ¿Conoce la deuda técnica de no tener el proyecto respaldado con pruebas unitarias, de integración y de validación para garantizar el buen funcionamiento del sistema a medida que se le introducen cambios? Más simple todavía, ¿entiende la enorme dificultad que supone siquiera estimar una tarea concreta y que cualquier estimación que hagamos siempre va a ser una aproximación más o menos real?

Un manager está obligado a conocer en profundidad la metodología que se emplea. Esto suena a perogrullada aunque me viene a la cabeza cierto proyecto en el que los *stand-up meetings*[7] terminaban durando más de una hora (¡cada día!). ¿Cómo se puede intentar mantener la disciplina metodológica de un grupo de trabajo si no es conocida en profundidad?

Estamos hablando de un responsable de equipo que trabaja directamente y codo con codo con los desarrolladores y, por tanto, debe conocer todas las circunstancias que afectan negativa o positivamente al trabajo de estos; el conocer estas circunstancias es su rol y su responsabilidad, ya que de ello va a depender directamente el resultado del trabajo en común. Un componente esencial es que un manager sepa que desarrollar software, aun considerándose lógicamente un trabajo como muchos otros, tiene un componente artístico y creativo continuo y requiere de cierta tranquilidad para que los desarrolladores trabajen con fluidez y concentración. Un desarrollador toma decisiones de diseño constantemente al igual que no puede bajar la guardia a la hora de generar pruebas de calidad.

Creo firmemente que un programador profesional siempre intenta hacer su trabajo lo mejor posible aunque para ello debe contar con cierta «paz mental» precisamente por los elementos que comentaba anteriormente. Su responsable es el encargado de generar en el equipo esa paz mental necesaria para que se haga un buen trabajo; parece que cuando hablamos de ese relativo sosiego que nos debe acompañar la mayor parte de días laborales nos estamos refiriendo a algo místico; nada más lejos de la realidad: un desarrollador de software necesita de cierta tranquilidad y seguridad diarias para que su trabajo no se vea afectado. En realidad cualquier persona que realiza una

tarea creativa las necesita. En esencia, esto supone no sufrir continuas interrupciones, que te inviten a reuniones maratonianas no previstas, que te pidan continuamente ciertas tareas para antes de ayer, etc. Un ambiente libre de todos esos elementos es casi imposible, pero sería suficiente si al menos el ochenta o noventa por ciento del tiempo el desarrollador pudiera trabajar concentrado y con el foco puesto en su trabajo.

Un equipo de desarrollo que trabaja con estrés necesariamente va a producir un resultado peor que si se consigue un buen ambiente y un entorno creativo. La productividad es también esto, conseguir mejores resultados sencillamente al tener un mejor entorno de trabajo.

Efectivamente, nos pagan el salario igual por producir *algo*, pero que nadie se lleve a engaño porque las circunstancias determinan la calidad de lo que producimos. No vivimos en un mundo ideal y como seres humanos y emocionales que somos sufrimos muchos factores que nos hacen encarar el trabajo creativo que hacemos de una manera u otra.

En ocasiones el manager proyecta sus propias ansiedades, presiones e incertidumbres sobre el equipo que gestiona: un buen gestor nunca hace eso, sabe, como se suele decir, «parar los golpes» antes de que lleguen al equipo y distinguir quién tiene que cargar con qué y cuándo. No se trata de intentar vivir en una burbuja aislados del resto de cosas que acontecen en la compañía, sino de evitar que ciertos elementos afecten innecesaria y negativamente al equipo de desarrollo, sobre todo si esos elementos son responsabilidad de otros roles.

En una ocasión trabajé en un grupo en el que el responsable tenía un pie dentro y un pie fuera del departamento precisamente por los malos resultados que estábamos obteniendo; no es cuestión en este ejemplo de encontrar el responsable de esa situación sino de indicar el mal ambiente y ansiedad que existía en el grupo de trabajo porque este manager transmitía sus inseguridades a todos y cada uno de los miembros del equipo. Esta situación duró varios meses y tengo que reconocer que en un ambiente así es imposible intentar hacer el mejor trabajo posible cuando estás esperando que el reloj te indique la hora para volver a casa.

Uno de los mayores vicios de los malos gestores es no cumplir con la planificación pactada (si es que esta existe) y decidir a su voluntad el

contenido de tareas de alguien de un momento para otro, sobre todo cuando se originan cambios o solicitudes «imprevistas» que hace que el equipo de programadores tenga que trabajar saltando de una cosa a otra con una total falta de previsión. Nada peor para un programador que cambiar de contexto continuamente dejando tareas inacabadas: el retomarlas requiere de un esfuerzo extra que, sea todo dicho, es totalmente improductivo. Uno de los principios del desarrollo ágil es que *done means done*[8]: comenzamos una tarea y la cerramos «completamente»; esto aumenta la productividad en varios niveles.

La productividad se lleva muy mal con malos gestores de equipos.

Ya no hablamos de falta de requisitos claros, sino de que estos cambien tan a menudo que haga imposible la más mínima organización. Quizá sea un escenario extremo aunque el manager tiene que saber decir «no» a sus propios superiores o al mismo cliente cuando la situación no es sostenible. En muchas ocasiones esta forma improvisada de trabajar tiene detrás una inercia de años o sencillamente una ausencia total de que alguien marque claramente la forma de trabajo.

Otra obviedad está relacionada con los equipos y herramientas que necesitamos para trabajar: los programadores no hacemos un trabajo de papel y bolígrafo (bueno, a veces sí, pero como medio para comenzar a plantear cosas con dibujos, esquemas, etc.); necesitamos por lo general un equipo hardware suficientemente dimensionado y a la medida del tipo de proyecto en el que estamos trabajando. Muy a mi pesar, he tenido varias experiencias en las que se llegó al punto en que compilar una solución completa tardaba más de cinco minutos en mi estación de trabajo. Esto es realmente frustrante; se podían contar por horas los tiempos muertos a lo largo de una simple semana.

En otra ocasión probar una solución era extraordinariamente complejo por las reglas de seguridad de la red de la compañía. ¡Cuántas horas perdidas por ese tipo de problemas!

En entornos ciertamente complejos, seguramente hagan falta más equipos para probar despliegues cuando no un sistema suficiente si tenemos implantado un entorno de integración continua.

Todo esto resume con el mayor sentido común que un equipo de desarrollo necesita los medios adecuados ¡para trabajar! El responsable del

equipo es quien tiene que detectar esas necesidades y proporcionarlas, no hay más. La buena productividad de un equipo viene determinada por contar con el hardware, herramientas, conexiones, etc. necesarios para que el trabajo se haga con fluidez y con los mínimos obstáculos; el gestor del equipo debe reducir éstos al mínimo posible.

En mi paso por una «gran» compañía tenía que conectarme continuamente a una VPN (red privada virtual) de un cliente en el extranjero. Para conseguirlo teníamos que hacer auténticas piruetas desde nuestro ordenador por «restricciones de seguridad» corporativas, o al menos eso nos decían cada vez que intentábamos mejorar la situación. El resultado era un entorno muy incómodo, lento y engorroso para resolver problemas con el cliente. Esta pérdida de productividad se tenía que traducir necesariamente en costes extra y un resultado de peor calidad.

Cualquier trabajo que tenga que hacer una persona, si se quiere hacer bien, necesita de un mínimo de organización y planificación; si tenemos un equipo de gente para realizar ese trabajo, para que este fluya correctamente, esa organización deberá ser aún mejor y si además el equipo es multidisciplinar con roles diferenciados, el no contar con la correcta planificación puede provocar un auténtico caos.

Muchos de los gestores de proyecto con los que he trabajado no han sabido organizar y planificar correctamente; esta es mi traumática experiencia y me gustaría pensar que no ocurre así en todos sitios. En muchos proyectos podríamos haber hecho un trabajo muchísimo mejor (y en menos tiempo) si hubiésemos contado con una mejor organización «desde arriba». Yo siempre digo lo mismo: cuando las horas extras pasan de ser puntuales a constituir algo crónico y el modus operandi del día a día, casi siempre detrás existe una mala organización y planificación.

Para que un equipo de programadores trabaje con la suficiente productividad (que no es trabajar más sino mejor) necesita funcionar como una máquina de engranajes perfecta: esta organización la debe proveer el responsable del equipo, pero para ello se requiere trabajar duro organizando y planificando, no «ordenando».

Aunque todo es siempre mejorable, actualmente trabajo con un equipo de cinco desarrolladores y dedico aproximadamente entre un veinte y treinta por

ciento de mi tiempo a organizar el equipo y a planificar la evolución de la plataforma que se está desarrollando. Me gustaría pensar que el buen ritmo al que progresamos y la suficiente calidad que estamos consiguiendo tienen que ver precisamente con ese tiempo que dedico a lo que es la mayor responsabilidad de un gestor de proyectos: organizar y planificar. Espero tener éxito en este empeño.

Gestionar significa conocer en profundidad lo que ocurre en el seno de un grupo de trabajo, ¿cómo si no vamos a tomar decisiones si no sabemos el progreso real, los problemas del día a día y las vicisitudes del grupo de profesionales que trabajan para obtener un buen resultado? Preguntar es bueno, aunque se da la circunstancia de que muchos gestores de proyectos software preguntan tanto que continuamente están incordiando y molestando al resto del equipo o bien están acostumbrados a convocar reuniones espontáneas sólo para resolver sus propias dudas; esta es la actitud de quien cree que el equipo «por debajo» de él está a su servicio, quizá por un mal entendimiento de las jerarquías, cuando en realidad es el manager quien debe estar al servicio del equipo para que este trabaje en las mejores condiciones.

Cualquier trabajo medianamente complejo se tiene que simplificar y tener bajo control con las herramientas adecuadas; un gestor de proyectos software debe contar igualmente con herramientas que le indiquen cómo está todo en cada momento sin necesidad de andar preguntando continuamente o llamando por teléfono a uno o a otro. Si no se tiene algún tipo de herramienta de gestión (que además las hay gratuitas con la suficiente calidad), ¿cómo si no vamos a detectar desviaciones importantes, cuellos de botella en tal área, etc.? Si no tenemos toda la información relevante en la gestión de un proyecto de alguna manera automatizada, no podremos anticiparnos a los problemas. Un manager debe anticiparse a ellos y trabajar para minimizarlos; para ello debe usar las herramientas adecuadas.

Ahora bien, ¿y los desarrolladores? No podemos tener derechos sin deberes del mismo modo que no podemos exigir que todo se nos dé hecho: tenemos la responsabilidad también de favorecer el buen trabajo del equipo y eso también supone ayudar y facilitar la labor del responsable del mismo. Los desarrolladores también deben permitir la correcta fluidez del trabajo que realiza el resto del grupo; el gestor del proyecto es uno más en él, ni más ni

menos importante, pero si está presente, entonces debemos favorecer su trabajo con la transparencia y honestidad que caracterizan a un buen profesional. Nada peor para un equipo de trabajo que un «troll» que continuamente boicotea la buena marcha del trabajo de los demás. Esto es también es un problema que hay que resolver.

Muchos proyectos fallan estrepitosamente por la ineptitud, falta de profesionalidad o conocimientos de quien es responsable de gestionar el equipo. Los programadores de software rara vez viven en entornos aislados sino que pertenecen a un grupo con el que se deben coordinar. Cuanto mejor funcione el grupo, todos y cada uno de sus miembros, mejor podremos realizar nuestro trabajo y mejor aún será el resultado global.

Si un proyecto software fracasa en cualquier sentido que entendamos por fracaso, el principal responsable es el gestor; no hay que darle más vueltas. Precisamente su papel es el de mover las piezas, generar las circunstancias y detectar los obstáculos para que el proyecto común termine con éxito.

Puntos clave

La actividad de un mal gestor de proyecto se detecta fácilmente; en *El Libro Negro del Programador* nos centramos en todos aquellos aspectos que afectan negativamente al desarrollo de un buen software; podemos decir que un producto de calidad es directamente proporcional al buen hacer del manager que lo ha dirigido.

Un mal gestor:

- No implementa una organización y planificación claras.
- Permite o provoca que se realicen tareas imprevistas con bastante frecuencia, cuando debería ser algo muy puntual.
- No impone ningún tipo de herramienta de gestión de proyectos (o bien lo que usa es una hoja de cálculo…).
- Convoca reuniones espontáneas como modo de trabajo habitual. No estoy en contra de alguna reunión no prevista por razones de urgencia pero sí de que se convoquen este tipo de reuniones constante y habitualmente.

- Crea incertidumbre y ansiedad en el equipo de desarrollo al trasladar sus propias inquietudes y ansiedades.
- No conoce en profundidad la metodología de desarrollo utilizada. El mal gestor de proyecto «cree» que eso no está en su función.
- Es incapaz de mantener la necesaria «disciplina» metodológica en todos los artefactos que se deben generar. Nada peor que saltarse la metodología en momentos de crisis.
- No se preocupa de mejorar el «ambiente de grupo».
- Acepta cualquier cambio radical de requisitos y modificaciones en la planificación sin ninguna oposición.
- No desprende la suficiente autoridad (que no «autoritarismo») para que sus decisiones sean respetadas.

El día a día de un buen desarrollador

«Un buen desarrollador de software trabaja con disciplina y constancia desde el primer día. Una mente ingenua y con poca experiencia pensará que esto significa trabajar duro e intensamente a cada momento. No es fácil ver y apreciar que esta disciplina no va a hacer que trabajes más, sino todo lo contrario: al trabajar con orden realizas un trabajo de manera más relajada, asegurando la calidad de lo que vas haciendo y llegando a las fechas previstas con comodidad. Esta manera de enfocar el trabajo se ajusta más a la naturaleza creativa del software.»

Como en muchas otras profesiones, uno de los peores defectos que podemos cometer quienes nos dedicamos a crear soluciones software es la inconstancia: sabemos que lo que tenemos entre manos debemos realizarlo para cierta fecha que al principio siempre parece muy lejana, de modo que cuando comenzamos es demasiada la tentación de trabajar relajadamente. A medida que pasa el tiempo nos damos cuenta de que se nos acumulan muchas tareas y comenzamos a trabajar con mayor ahínco y dedicación. Pronto aparecen el agobio, las prisas y las horas extras, al tiempo que comienza a resentirse la calidad del trabajo que realizamos.

Que conste que soy el primero que he caído en la tentación de esta manera de trabajar en muchas ocasiones, aunque también por eso he podido analizar las consecuencias a largo plazo sobre la productividad, la satisfacción personal y la validez del trabajo realizado.

Esta falta de disciplina y de constancia en el tiempo, el esfuerzo y la concentración en el trabajo desarrollando software tiene resultados distintos

que cuando te dedicas a otro tipo de actividad: el que fabrica tornillos terminará fabricando menos, el que repara vehículos le tendrá que pasar más horas a la factura del cliente, el que pinta un chalet (por poner un ejemplo) tardará más en terminar el trabajo aumentando la probabilidad de ganar un cliente insatisfecho. En ese tipo de trabajos por horas lo inevitable será que pierdas dinero porque has tardado más en entregar el producto, sean tornillos, un vehículo reparado o la casa de tu vecino pintada y reluciente. Cualquier trabajo hecho con prisas siempre se hace peor (pero ¡qué sorpresa!).

No obstante, en software, esta falta de constancia afecta radicalmente a la calidad interna del trabajo entregado; el producto funcionará, seguramente (aunque aumenta la probabilidad de entregarlo con errores), pero se habrán cometido la mayor parte de los pecados capitales del software que nos pasarán factura más adelante en forma de numerosos «*to do…*» no implementados, pruebas por hacer, código de tipo espagueti, etc.

La inconstancia en el trabajo de un programador se paga con un aumento de la deuda técnica (todo eso que vamos dejando y que termina explotando mucho más adelante, a nosotros o a algún compañero).

Un buen desarrollador de software trabaja con disciplina y constancia desde el primer día. Una mente ingenua y con poca experiencia pensará que esto significa trabajar duro e intensamente a cada momento. No es fácil ver y apreciar que esta disciplina no va a hacer que trabajes más, sino todo lo contrario: al trabajar con orden realizas un trabajo de manera más relajada, asegurando la calidad de lo que vas haciendo y llegando a las fechas previstas con comodidad. Esta manera de enfocar el trabajo se ajusta más a la naturaleza creativa del software.

Lo he visto (y sufrido) una y otra vez y de nuevo tengo que decir que es un error pensar que quien dedica más horas (o quien está más tiempo presente en la oficina) es quien más trabaja, cuando en cambio esto debe interpretarse precisamente como un síntoma de falta de productividad y eficiencia en el día a día. Esta falta de productividad puede ser personal o de la misma organización. Teniendo en cuenta que suponemos que un profesional siempre quiere hacer su trabajo lo mejor posible, los problemas de organización y de planificación suelen ser de la compañía. Otro asunto es que el trabajo de desarrollo de software no se puede medir por horas, sino por resultados. Este

es desde luego un tema muy amplio pero baste decir aquí que cuando alguien «necesita» continuamente estar más horas de su jornada laboral para terminar sus tareas, entonces estamos ante algún tipo de problema de organización.

En nuestro currículum académico no se menciona en ningún momento una de las leyes que acompañan el desarrollo del buen software: no avanzamos en nuestro trabajo mejor cuando dedicamos intensos y discontinuos ciclos de desarrollo (hoy aprieto y mañana ando un poco más relajado) sino cuando somos capaces de mantener una relativa constancia y la suficiente intensidad creativa en todo momento. Ya sabemos que el estrés y le presión van en contra de los buenos hábitos de un programador, por tanto la táctica consistirá en evitarlos al máximo.

Podemos resumir con un ejemplo los hábitos más importantes en el día a día de un buen desarrollador: tenemos dos semanas por delante para implementar una solución y está definido perfectamente lo que se tiene que hacer, no hay dudas al respecto. Hay, no obstante, algunas lagunas sobre cómo implementar ciertas partes.

Un buen desarrollador identifica claramente las piezas fundamentales de la solución, comenzando a abordar aquello que «aún no sabe» cómo implementar o cómo darle forma, esto es, comenzamos por aquello que más nos incomoda. Esto es fundamental: si terminamos antes lo que más nos inquieta, eliminaremos rápidamente el ronroneo mental de tener aún cosas incómodas o difíciles que afrontar.

Se realiza algún *spike* de código para probar la mejor manera de hacer esto o aquello. Cuando ya lo tenemos, estamos listos para integrarlo en la solución general. De paso nos quedamos completamente tranquilos porque ya hemos solucionado lo que más nos preocupaba del desarrollo. Así, solucionamos al principio esta inquietud, no la acumulamos en los últimos días de las dos semanas cuando otros problemas nos vendrán a agobiar más.

Se realiza una de las partes de la solución y se valida suficientemente con pruebas, unitarias, de integración, etc. No entraremos aquí si usamos o no TDD (*test driven developement*), este no es un libro técnico, pero sí dejamos constancia de que hay que probar el módulo o la parte que estás haciendo para garantizar que más adelante, cuando la solución avance y evolucione, esta parte en concreto seguirá funcionando. Es más, hay que automatizar esas

pruebas para que se puedan ejecutar una y otra vez: en definitiva te van a decir que todo lo que tienes hecho funciona correctamente.

Continúas con un segundo módulo en donde comienzas a ver ciertas similitudes con el primero: no lo duplicas, sino que dedicas una hora a ver en qué consisten estas similitudes y extraes funcionalidad común: abstraes algo que anteriormente no habías previsto. Esto requiere un esfuerzo y seguramente tocar el primer módulo. Al modificarlo, tendrás al mismo tiempo que cambiar algunas de sus pruebas. Aquí chocamos con uno de nuestros mayores defectos: la aprensión a modificar lo que ya sabemos que funciona, como ya hemos comentado en detalle en otro capítulo de *El Libro Negro del Programador*. No se retoca porque sí, sino para «mejorar la solución completa», por tanto estamos invirtiendo tiempo y esfuerzo en algo que nos reportará beneficios más adelante.

Terminamos el segundo módulo y antes de comenzar el tercero (ya ha pasado la primera semana), ahora que conocemos mejor cómo vamos a resolver la solución, nos preguntamos qué hay de lo ya desarrollado que se pueda mejorar o simplificar.

Esta pregunta es fundamental, porque si la incluimos en nuestro ADN de desarrolladores profesionales, en realidad lo que estamos implantando es una mejora continua en nuestro trabajo: de nuevo construimos de abajo hacia arriba, cuando mejor funcione lo primero mejor resultará lo segundo. El mejorar en algo estos dos primeros módulos nos reportará beneficios de aquí en adelante.

Es sorprendente a veces el poco tiempo que necesitamos para simplificar y mejorar algo. En ocasiones es sólo cuestión de minutos: cuando nos demos cuenta de que podemos hacer algo mejor en el mismo tiempo que dedicamos a leer algunos *tweets* intrascendentes (por poner algún ejemplo) entonces habremos dado un salto de nivel. Con mucha frecuencia las grandes mejoras de una solución no vienen dadas por un gran replanteamiento de su estructura, organización, etc. sino de la acumulación durante mucho tiempo de pequeñas mejoras incrementales.

Volviendo a nuestro ejemplo, una vez hecho el cambio, ¿cómo estamos seguros de que ambos módulos siguen funcionando bien? Gracias a que hemos apoyado suficientemente con pruebas el desarrollo que tenemos hasta

ahora, se vuelven a ejecutar y comprobamos que todo sigue funcionando correctamente. Seguramente esta última simplificación nos habrá obligado también a cambiar y mejorar algunas pruebas; este trabajo no es un gasto sino una inversión, ya que con él estamos garantizando que nuestra solución funciona y disponemos de un método para comprobar que seguirá funcionando a medida que se vaya introduciendo más funcionalidad.

Así afrontamos el tercer módulo, con la disciplina de un trabajo constante y con la pregunta recurrente ¿puedo mejorar o simplificar algo? y con la disciplina de ejecutar continuamente las pruebas para detectar rápidamente bugs imprevistos.

Me pasa en ocasiones que sufro de un ronroneo mental continuo cuando hay algo que debo hacer en algún momento de la semana, por poner un ejemplo, pero que es un poco tedioso, rutinario o sencillamente un problema que llevo un tiempo arrastrando pero que hay que resolver. Sin darte cuenta, continuamente en tu mente saltan alertas de tipo «tienes que hacer esto y aquello que no te gusta nada en absoluto». Llega por fin el momento en el que decido hacerlo y para mi sorpresa, aquello que me había estado persiguiendo tanto tiempo lo liquido en un suspiro. Entonces, me pregunto siempre, ¿por qué no lo hice hace tiempo y «me lo quité de encima»?

Sin darnos cuenta, nuestro trabajo seguramente esté jalonado de esas pequeñas tareas insidiosas que tenemos que realizar queramos o no; si se hacen rápido, mejor hacerlas pronto y evitar que la mente te lo recuerde continuamente, así nos dedicamos con más calma al resto de cosas.

Un consejo que propone David Allen, en su magnífico libro «*Getting Things Done*» (que en español ha sido titulado como «Organízate con Eficacia») en el que describe el método de organización de trabajo GTD, es que siempre que haya algo que se pueda resolver en menos de dos minutos, hay que hacerlo al instante. Nos evitamos divagaciones mentales recordándonos continuamente que tenemos algo pendiente; esas divagaciones nos impiden concentrarnos en lo que realmente importa más.

Este proceso de trabajo continuo que hemos descrito en el ejemplo anterior, aunque muy simplificado, es el que nos permite terminar las dos semanas con el trabajo completado y con la seguridad y satisfacción de que llegamos a tiempo demostrando que lo que hemos hecho funciona

correctamente ya que está respaldado suficientemente con pruebas y con una enorme satisfacción interior por haber hecho un buen trabajo. En el libro «*The Passionate Programmer*»[9] (de Chad Fowler) se hace referencia muy acertadamente a la necesidad de encontrar esa satisfacción feliz en nuestro trabajo, al que, por cierto, le dedicamos en el día a día mucho más tiempo que a cualquier cosa; por tanto, ¿por qué acostumbrarnos a gastar mercenariamente tanto tiempo de nuestra vida a algo que «sólo» nos va a reportar un salario? Tenemos la obligación de buscar una actividad satisfactoria y ser felices con lo que hacemos; personalmente, puedo decir que una de mis mayores satisfacciones es ver terminado un buen trabajo.

Este proceso creativo se ve interrumpido o es imposible de llevar a cabo si no mantenemos mínimamente esa disciplina diaria, ese mirar atrás constante que nos permite identificar qué mejorar y qué simplificar.

El trabajo en el día a día debe consistir siempre en una constancia que nos permitirá construir sólidamente las partes de la solución. Si corremos, lo primero que perderemos serán las pruebas tan necesarias para comprobar que todo funciona correctamente a medida que introducimos cambios, si dejamos que pase el tiempo y nos ponemos a trabajar duro los últimos días de la entrega nos arriesgamos a estrellarnos y seguramente lo que se haga deprisa y mal habrá que volver a rehacerlo más adelante (pan para hoy y hambre para mañana).

Recientemente, en el momento de escribir esto, afronté un desarrollo de algo completamente nuevo en un *sprint* de tres semanas: en la tercera poco de lo que había hecho en la primera había quedado sin tocar, las pruebas no tenían nada que ver con las que hice durante los diez primeros días, el diseño había emergido de manera natural y al final tenía una solución con la garantía de que funcionaba (al menos pasaba todas las pruebas) y la satisfacción y ganas de seguir evolucionándola.

Honestamente, ¿estamos cómodos cuando tenemos que retocar una solución quebradiza y que se rompe al más mínimo cambio o nuevo requisito? Es mucho más probable que podamos y queramos evolucionar un software elegante, bien diseñado, claro y sencillo.

¿Entendemos por tanto que esa disciplina de la que hablamos nos hace llegar al segundo escenario? En definitiva, el trabajar de ese modo, más

ordenado y con ánimo siempre de mejorar, nos permitirá mañana disfrutar más de nuestra labor como desarrolladores de software.

Es mucho más importante y mejor trabajar con constancia en el día a día comenzando por las cosas que más nos inquietan que abandonarlo todo a que nos entren ganas de desarrollar esto o aquello con la presión del tiempo apuntándonos detrás de la nuca. Nunca podremos hacer un bueno trabajo con una presión excesiva y estrés continuado.

Puntos clave

- Siempre debemos comenzar una fase de trabajo por aquello que más nos inquieta o lo que consideramos más complicado; al eliminar esta preocupación, afrontaremos el resto de tareas más relajadamente.
- Debemos progresar en la solución con constancia: nada peor para la calidad del software que alternar ciclos muy intensos de trabajo con otros de inactividad.
- Un buen desarrollador siempre «mira hacia atrás»: busca en aquello que ya se ha implementado qué se puede abstraer, mejorar o simplificar. Eso es así porque vamos aprendiendo la mejor manera de construir una solución ¡construyéndola!

El éxito siempre viene de la unión entre Talento, Tecnología y Metodología

«El fracaso de un proyecto siempre está determinado por la mala combinación de esos tres conceptos: talento, tecnología y metodología; o lo que es lo mismo, el éxito de un proyecto siempre cuenta con esos tres factores bien equilibrados, pero los tres a la vez y sin la ausencia de ninguno de ellos.»

Recientemente me pidieron que evaluara el estado de un proyecto software, una mezcla de ERP, CRM, etc. El cliente, en este caso el dueño y gerente de la compañía quería conocer el grado de avance del equipo de desarrollo interno responsable del mismo.

La preocupación era evidente: pasaban las fechas previstas y no tenía narices de ver algo que le permitiera suponer que se estaba avanzando en un proyecto que, en definitiva, él estaba financiando y que iba a ser vital para el desarrollo futuro de la compañía.

El equipo de desarrollo, por su parte, se mostraba reacio a mostrar nada y ni siquiera eran capaces de ofrecer una fecha estimada de entrega de la solución. Me pregunto si en software tiene sentido o no «entregar» algo terminado y cerrado a cal y canto. La frustración e inquietud del primero era proporcional a la incertidumbre del equipo de desarrollo: una situación muy complicada, tensa y desagradable para todos los involucrados.

Me bastó una hora de charla con el responsable de la solución para detectar varios de los males endémicos de nuestra profesión. Veinte minutos de mirar el código fueron suficientes para descubrir muchos «debes» que clamaban al cielo; menos tiempo aún necesité para sacar ciertas conclusiones

al ver las «evidencias» que el equipo de desarrollo había ido dejando en la herramienta de gestión de proyecto. Demasiadas similitudes con otros proyectos en los que yo mismo había participado. ¿Por qué tanta gente tropezamos siempre con las mismas piedras?

No me resulta nada agradable cuando me ponen en situación de evaluar el trabajo de un compañero que se gana la vida haciendo lo mismo que tú, con mayor o menor acierto; presuponiendo la buena voluntad y buena disposición hacia el trabajo, es muy incómodo detectar (y comunicar) las insuficiencias y carencias. Parto de la base de que todo el mundo hace lo que puede y en muchísimas ocasiones no se nos dan suficientes condiciones necesarias para avanzar en el trabajo propuesto de manera correcta y con calidad.

Este es un caso concreto con el que me he encontrado en el último año pero que sin embargo es paradigmático en un gran porcentaje de proyectos software: la mayoría de los que fracasan, lo hacen por tener una combinación no equilibrada de talento, tecnología y metodología. Pero, ¿qué entendemos por estos tres conceptos para el éxito de un proyecto?

No hace falta ser ningún gurú (¿he dicho ya que los gurús o «máquinas» de la computación casi nunca sirven para trabajar en un grupo de desarrollo?), ni ningún *Bill Gates* ni nada por el estilo: el sentido común se impone. Me pregunto por qué no hay clases durante nuestra formación de sentido común para informáticos. El éxito en la consecución de un proyecto software, más o menos complejo, que involucra a un equipo de desarrollo con roles diferenciados, no está en un único factor determinado, sino en varios y éstos tienen que estar suficientemente equilibrados.

Algunos miembros del equipo pueden ser geniales, súper listos y desplegar talento por los cuatro costados; les gusta todo lo que hacen así como investigar nuevas vías para hacer y mejorar lo que ya funciona. Son los que causan el efecto «guau, a mí nunca se me habría ocurrido»; van tan rápido en su forma de causar asombro que no tienen tiempo de documentar mínimamente lo que hacen (total, si ya es genial, para qué), tampoco les gusta que nadie les ponga encima varios «posits» con las tareas que deben hacer para los próximos días porque eso puede coartar su creatividad. En cuanto leen un artículo con un trabajo estupendo de lo que yo llamo «ingeniería-fontaneril» (o sea, de «tocar las tripas de»), rápidamente sienten

la necesidad de aplicarlo a lo que tienen en ese momento entre manos «de la manera que sea», aunque entre con calzador. Están, digamos, viciados por su propia genialidad, despliegan un talento avasallador, pero no les pidas que centren el tiro, se ciñan a los estándares tecnológicos definidos y que den visibilidad de lo que están haciendo en cada momento a su manager, jefe, *scrum master* o lo que sea.

Su genialidad y destreza les hacer ser imprevisibles y trabajar de manera muy improvisada.

Por otro lado, otros miembros del equipo van a un ritmo, digamos, menos acelerado, les cuesta más entender lo que tienen que hacer, meten la pata recurrentemente y para ellos la proactividad es una palabra demasiado compleja. Hacen lo que pueden, tutorizados siempre por alguien de modo que no se les puede pedir que hagan demasiados inventos ni que tengan demasiada iniciativa.

Unos y otros son los extremos contrapuestos del «talento». Yo siempre asumo que la gente es mucho más inteligente de lo que aparenta y que existen verdaderos diamantes en bruto que ni ellos mismos pueden sospechar que lo son. La falta de talento no suele coincidir en una merma de recursos intelectuales… sino de una falta de motivación, actitud, etc.

Por otra parte, algunos miembros del equipo tienen un dominio de la tecnología que asombra: les basta con ver el síntoma de un problema para detectar dónde está la causa, saben qué trabajo de fontanería tienen que tocar para conseguir un resultado, les gusta eso de *cuanto-mas-intrincado-e-interno-mejor*.

En cambio, otros, *googlean* continuamente cada vez que tienen que implementar una clase abstracta y no saben por qué protesta el compilador, buscan en foros ese nuevo mensaje del IDE que les dice que hay dependencia circulares o ante el problema «mostrar por orden alfabéticamente la lista List<>; de» buscan por «mostrar por orden alfabéticamente una lista List<>». Son, en definitiva, gente despierta pero para «otras cosas». O a lo mejor, estaban más hechos al mundo Java donde sí se movían con extrema facilidad pero ahora se han visto obligados a usar C#, por poner un ejemplo.

De nuevo estamos ante el ying y el yang del mismo concepto: «el dominio o no de la tecnología» que se ha decidido usar. No hay que

confundirlo con el talento sino con conocer suficientemente bien las herramientas, frameworks, arquitecturas, etc. necesarias para el trabajo.

De nada nos sirve tener el suficiente talento y conocer suficientemente bien las herramientas y tecnologías si no tenemos un marco de trabajo ordenado que nos permita avanzar en equipo.

Esta cuestión es clave y se sintetiza en las siguientes dos preguntas: ¿se ha decidido seguir una metodología en el desarrollo de la solución?; si a esta pregunta se responde con un «sí», entonces viene la segunda, ¿pero de verdad se sigue con igual intensidad y fuerza esa metodología a lo largo de toda la vida del proyecto? Ya el tema en sí de plantear o no el usar una metodología tiene sustancia y contenido para un libro aparte…

Esto es como los coleccionables después del verano o los propósitos bien intencionados del inicio de año: se confunde en ocasiones las intenciones y deseos con la realidad, ya que nos jactamos de seguir tal o cual metodología pero no reconocemos que comenzamos muy bien, después flojeamos y cuando tenemos la más mínima presión de tiempo, lo primero que abandonamos es ¡la metodología!

Sobre esto hablamos largo y tendido en *El Libro Negro del Programador*, pero dejadme recordaros que existe una realidad muy sutil sólo al alcance de iniciados: cuando la presión se presenta en un proyecto software, lo primero que muere es la metodología, es su primera víctima, ¡pero es que seguramente es la falta de metodología lo que ha provocado esa misma presión! Entender esto requiere de cierto esfuerzo introspectivo cuando fracasan los proyectos y se analizan las causas de esos fracasos.

Implantar una metodología sin la disciplina de aplicarla día a día es no seguirla, así de simple. En ocasiones me he encontrado más interés en decir y publicitar que «seguimos tal o cual metodología» que en aplicarla realmente con todas sus consecuencias.

Por tanto, vemos que hay tres conceptos que aplican a nuestros desarrollos de software y a continuación nos preguntamos cómo estamos en cada uno de ellos: ¿tenemos talento?, o, ¿la gente percibe que tenemos talento para lo que nos dedicamos?, ¿dominamos la tecnología que se ha decidido usar?, ¿el manager o gestor del proyecto cuenta con la disciplina suficiente para implantar la metodología con igual fuerza en el principio, el durante y el

final del proyecto?, ¿nos ceñimos en nuestro rol a esa metodología en todo momento? ¿Contamos con la necesaria formación y recursos para usar la tecnología del proyecto suficientemente bien?

Veremos que las respuestas a estas preguntas, si parten de la sinceridad, nos darán un «sí, sí, por supuesto en esto y aquello pero no en lo otro», o «pues a mí es que eso de documentar…» o «lo mío es codificar, entro en estado de fluidez y se me pasan las horas volando, pero es que las reuniones de control de proyecto me matan, para mí son pérdida de tiempo», y un largo corolario de posibles combinaciones.

El fracaso de un proyecto siempre está determinado por la mala combinación de esos tres conceptos: talento, tecnología y metodología; o lo que es lo mismo, el éxito de un proyecto siempre cuenta con esos tres factores bien equilibrados, pero los tres a la vez y sin la ausencia de ninguno de ellos.

Esto uno tarda años en darse cuenta y lo que es peor, apenas lo tienen en consideración los equipos gestores a la hora de organizar un proyecto. Uno de los fallos fundamentales de un gestor de proyecto es desconocer la fortaleza o debilidad de estos tres aspectos en el equipo que gestiona.

De nada nos sirve tener un talento radiante y natural en resolver problemas software si no dominamos suficientemente la tecnología subyacente que usamos, de poco nos sirve dominar hasta extremos esta tecnología si no tenemos la disciplina de seguir una metodología, la que se haya definido en el proyecto e igualmente, por mucha disciplina metodológica que sigamos, el proyecto no va a terminar bien si no somos capaces de hacer un módulo software sin errores.

¿Qué pasó en el caso que comentaba al principio, aquel proyecto que me pidieron que evaluara?

En ese lamentable caso, fallaron los tres pilares del éxito de un proyecto software: el talento era insuficiente, en forma de un equipo sin la necesaria experiencia en un proyecto similar (no digo que fuesen tontos, ni muchísimo menos, eran y son tíos muy listos y competentes), el dominio de la tecnología a usar era mejorable (en ese caso .NET framework y tecnologías relacionadas, Entity Framework, etc.) y, para colmo, se comenzó el proyecto con mucha fuerza siguiendo SCRUM pero ésta se fue diluyendo poco a poco

(a las pocas semanas más bien), al no percibir claramente la rentabilidad metodológica de aplicarla disciplinadamente.

El resultado de todo esto no es sólo un proyecto que fracasa y un equipo frustrado que realmente no termina de entender qué se ha hecho mal y que en realidad ha trabajado con la mejor voluntad del mundo, además hay un agujero económico que alguien tiene que cubrir.

Puntos clave

- Muchos proyectos software fracasan porque no ha existido el correcto equilibrio entre talento, tecnología y metodología.
- En mi humilde experiencia, se suele disponer de mucho talento y conocimiento de la tecnología y casi siempre lo que falla es la correcta implantación de una metodología, cualquiera que sea esta.
- Es preferible tener la suficiente combinación de esos tres aspectos que contar con auténticos «gurús» o llaneros solitarios incapaces de trabajar correctamente en equipo.
- El responsable del equipo debe evaluar estos tres factores y detectar fortalezas y debilidades (aunque reconozco que esto es siempre mucho más fácil de decir que de hacer).

El mal entendido rol de arquitecto de software

«La presencia forzada de un arquitecto de software que obliga a ceñir la solución a una arquitectura diseñada al principio es más un obstáculo que un elemento positivo en el proyecto. He podido comprobar (y sufrir) cómo en ciertos proyectos los desarrolladores trabajaban más y dedicaban más tiempo a meter con calzador algo en la arquitectura establecida que en hacer software útil. Este problema es muy difícil de resolver porque precisamente la presencia de un super-arquitecto-software revela la existencia de roles jerarquizados: un obstáculo total al desarrollo democrático y horizontal de un proyecto.»

Hubo una ocasión en la que participé en un gran proyecto internacional con un grupo amplio de desarrolladores implicados; la competencia profesional de estos era inmejorable, todos teníamos unas enormes ganas de afrontar los nuevos retos y además (cosa insólita) hasta contábamos con un grupo de testers desde el principio.

En *El Libro Negro del Programador* nos centramos en aquello que solemos hacer mal como desarrolladores y en las razones (casi siempre las mismas) por las que fracasan los proyectos software. En esta ocasión, me temo que la presencia de un *super-arquitecto-de-software* la vamos a describir más como un obstáculo que como un rol imprescindible.

Pues bien, el proyecto internacional que comentaba fracasó estrepitosamente al cabo de unos meses: con el tiempo se consiguió hacer funcionar lo que pudo haber sido un magnífico producto aunque para muchos se convirtió en una auténtica pesadilla y una etapa laboral realmente amarga.

Entre los muchos errores que se cometieron, el fundamental, a mi juicio, fue el de establecer claramente una persona con el rol de «arquitecto de software» para que a partir de sus decisiones de diseño y arquitectura fuese desarrollándose el universo software que debía constituir el proyecto y el éxito común del grupo.

Nada de esto sucedió, me temo, fundamentalmente por la mala concepción entonces de lo que debía ser un arquitecto. Aquí la clave está en haber planteado una arquitectura cerrada y determinada «al inicio del proyecto», cuando aún no se tenían claras todas las especificaciones y no se sabía cómo se iban a resolver las que sí se conocían a ciencia cierta.

A mi entender, una persona puede considerarse un «arquitecto de software» siempre y cuando reúna al menos las siguientes condiciones:

- Posee una experiencia altísima para un determinado «dominio» específico de trabajo. Esto es, nadie puede considerarse «arquitecto de software» para cualquier tipo de proyecto. Desarrollar bien supone conocer igual de bien el tipo de negocio para el que desarrollamos una solución.
- Ha pasado por muchos roles en su carrera (tester, desarrollador, analista, etc.). Esto viene a significar muchos años de experiencia con diferentes perfiles. Nadie termina su formación académica y rápidamente se convierte en un «arquitecto de software».
- Ha trabajado en productos de muy diversa y heterogénea naturaleza.
- No sólo debe tener un dominio total de las tecnologías que se usan sino que además tiene la capacidad de decidir cuáles son las mejores para la naturaleza del proyecto: un arquitecto de software no dice «usamos java *net beans* porque sí, porque a mí me gusta y punto» o «vamos a usar el último .NET framework porque… es lo último de lo último». Quizá este tipo de decisiones las deba tomar el líder técnico, pero poniendo los pies en el suelo y en la realidad, vemos que este tipo de cosas (la arquitectura, las tecnologías, el cómo y el por qué) la decide casi siempre una misma persona.
- Un arquitecto de software debe además saber liderar y guiar el trabajo de

> un grupo de desarrolladores. En la teoría podemos ver este rol como el de alguien que escribe sesudos documentos, pero en la práctica, si existe, quien posee este rol no es más que uno más de nosotros que toma decisiones. Además, un desarrollador de base necesita conocer las razones para las decisiones de diseño que él mismo no ha tomado. Nada peor que programar de un modo encajonado dentro de una arquitectura sin conocerla suficientemente.

En mi opinión muy pocas personas reúnen un perfil con estas características, entre otras cosas es incluso raro conocer a alguien que lleve más de diez o quince años programando, dada la frívola cultura empresarial según la cual ascender significa más bien gestionar dejando a un lado los asuntos técnicos…, problema del que hablamos extensamente en este libro.

Otra razón por la que es difícil reunir un perfil así es precisamente el carácter vocacional de nuestra profesión: si realmente te mantienes años y años mejorando y desarrollando frameworks, proyectos y productos es porque uno ama demasiado lo que hace. Cuánta gente he conocido que a los pocos años ha abandonado el arte de programar por otras actividades digamos, más sencillas. El desarrollo de software es quizá de las profesiones que más te obligan a reinventarte continuamente dada la enorme y rápida evolución de tecnologías, usos, herramientas, tipos de productos, etc. Como consecuencia de todo esto, pocos se mantienen al pie del cañón durante muchos años.

Sostengo en este libro, y además lo digo y afirmo con perfecto conocimiento de causa y experiencia, que en la mayoría de los proyectos no hace falta un rol de arquitecto de software: hablar de un perfil así es tanto como decir que el modelo de desarrollo en cascada sigue siendo el ejemplo a seguir.

A lo mejor me equivoco, pero a estas alturas de nuestra profesión, ha quedado demostrado que plantear una arquitectura «al principio» de un proyecto es un profundo y tremendo error. Insisto en lo de «la mayoría de proyectos», ya que en aquellos en los que intervienen múltiples agentes, por ejemplo, diversas compañías con distintas responsabilidades en el proyecto, sí es necesario establecer ordenadamente cómo y de qué manera las diversas

piezas del puzle van a encajar.

Por otro lado, no hay que confundir el rol de arquitecto con el de responsable o líder técnico, aunque en muchas ocasiones coincidan en la misma persona.

Salvo que tengas una experiencia muy pero que muy dilatada, cuando desarrollamos software desconocemos completamente cómo vamos a resolver ciertos aspectos de la solución hasta que realmente el problema lo tenemos de frente y de algún modo tenemos que salir adelante.

En otras ocasiones, lo que creíamos que iba a ser un paseo de rosas se convierte en un calvario por la cantidad de problemas no previstos. A menudo entendemos mal los requisitos, con mayor frecuencia aún estos han sido mal tomados, otras veces los requisitos están en un excel y además están obsoletos…

Estamos definiendo precisamente el tipo de problemas que surgen cuando entendemos el desarrollo de software como algo predictivo y previsible cuando por el contrario es algo más artístico e imprevisible de lo que pensamos. De aquí la necesidad de otro tipo de enfoque cuando resolvemos problemas software.

Y llegó el desarrollo ágil…

Existen tantas incertidumbres a la hora de programar algo que el único enfoque posible es hacerlo poco a poco, paso a paso. En lugar de entregar al final la solución completamente terminada (y muy seguramente incompleta según las expectativas del cliente), las entregas son cortas y frecuentes, asegurándonos que avanzamos con buen pie y sobre las expectativas reales del cliente.

En esto consiste básicamente el desarrollo ágil y todas sus variantes metodológicas.

Una de las características que más me gusta de lo ágil es que el diseño y arquitectura del sistema van surgiendo («emergiendo») a medida que se avanza en el desarrollo del mismo.

Al centrarnos en el desarrollo exclusivo de una parte o funcionalidad del sistema nos damos cuenta de que la arquitectura de la aplicación no es algo que debe ser pensada sesudamente al principio, sino que ésta surge espontáneamente a medida que se van implementando poco a poco los

requisitos establecidos.

Lógicamente, el avanzar en software de este modo nos obliga a hacerlo de otra manera: vamos preparando lo que desarrollamos de modo que sea fácil modificarlo y cambiarlo, vamos incluyendo funcionalidad asegurándonos en todo momento de que lo anterior no deja de funcionar y sobre todo haciendo *refactorings* continuamente.

Existe una abundante bibliografía dedicada al desarrollo ágil y no es cuestión de extenderme en este capítulo sobre algo que hoy día debería formar parte del acervo cultural de cualquier programador profesional, pero baste decir que al enfocar el desarrollo de esta forma, el rol de arquitecto de software se diluye; es, digamos, más prescindible.

La presencia forzada de un arquitecto de software que obliga a ceñir la solución a una arquitectura diseñada al principio es más un obstáculo que un elemento positivo en el proyecto. He podido comprobar (y sufrir) cómo en ciertos proyectos los desarrolladores trabajaban más y dedicaban más tiempo a meter con calzador algo en la arquitectura establecida que en hacer software útil. Este problema es muy difícil de resolver porque precisamente la presencia de un *super-arquitecto-software* revela la existencia de roles jerarquizados: un obstáculo total al desarrollo democrático y horizontal de un proyecto.

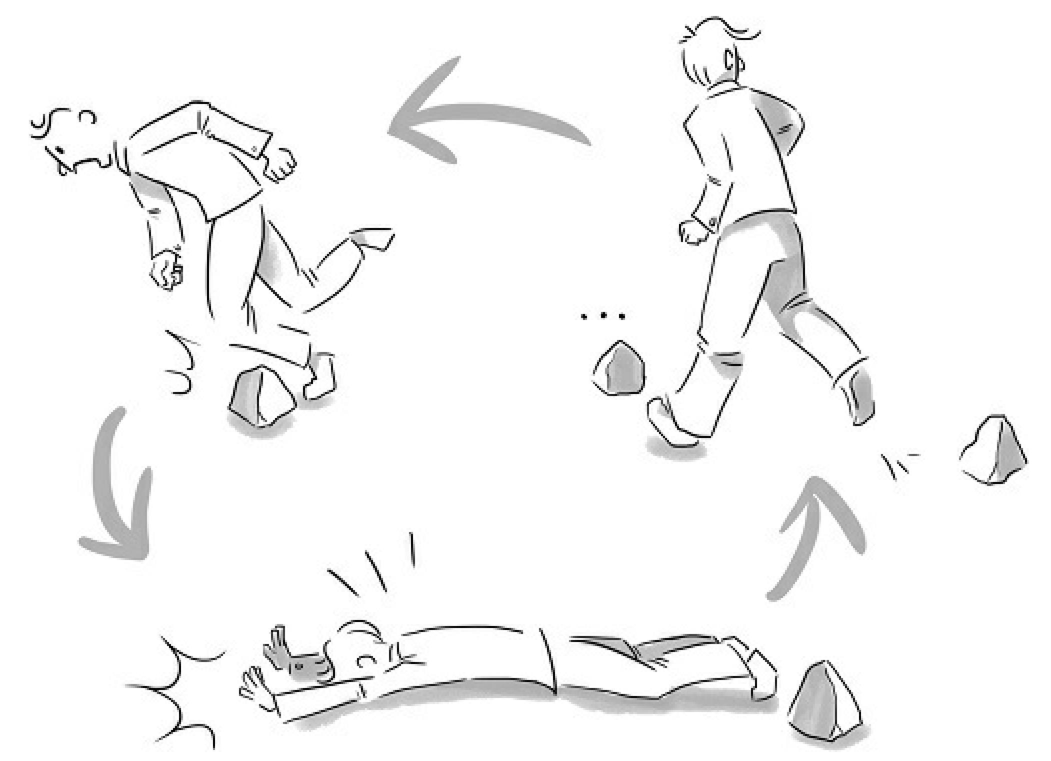
Esto tampoco significa que una vez dados los requerimientos nos debamos poner como locos a desarrollar, todo lo contrario: en el término medio es donde siempre residen las mejores decisiones. Cierto tipo de arquitectura elástica general como marco de desarrollo de la solución debe ser establecida, pero para esto no hace falta un rol tan cerrado como para ocupar al 100% el trabajo de una persona. Y es que me temo que también es una cuestión de ego… En mi modesta experiencia, cuando me he cruzado con alguien empeñando el rol de arquitecto el fracaso ha venido asegurado detrás: este rol se emplea más bien como jerarquía de poder dentro de un grupo de trabajo y de reconocimiento social que realmente como elemento favorecedor del desarrollo de un buen proyecto. La verdad es que suena mucho mejor «soy arquitecto de software» que «soy programador, desarrollador de software», aunque la dedicación y experiencia del segundo supere en varios niveles a las del primero.

También he visto cómo el desarrollo ágil ha tenido dificultades de penetración en grupos de trabajo precisamente porque con él llega la abolición de ciertos roles consideramos, digamos, más respetables.

En desarrollo ágil, la arquitectura emerge a medida que se avanza en la solución. En cierta medida, se desvanece así el rol de arquitecto de software. Esta idea fue en su momento tan revolucionaria que creo que sólo los que hemos visto fracasar proyectos por un diseño erróneo o una mala arquitectura la hemos abrazado completamente.

Puntos clave

- En la mayoría de proyectos no es necesario la presencia de un rol de arquitecto software.
- Sí es necesario en proyectos realmente grandes, con multitud de agentes implicados.
- Dada la progresión laboral de la mayoría de programadores profesionales, realmente pocos pueden llegar a reunir los requisitos necesarios para considerarse «arquitectos de software».
- El enfoque ágil diluye el papel de un arquitecto de software: la arquitectura «emerge» por sí sola a medida que se van implementado requisitos.



COMETEMOS LOS MISMOS ERRORES UNA Y OTRA VEZ

Los desarrolladores tienen en sus manos producir software de calidad y en las mejores condiciones siempre y cuando aprendamos proactivamente de aquellos errores que lastran nuestro trabajo y que la productividad es lo que nos permite centrarnos en lo verdaderamente importante, lo que aporta valor a una solución.

La rentabilidad metodológica

«Es difícil percibir que la metodología no sea una cosa más que se añade a las actividades propias de desarrollo, sino que forma parte intrínseca del mismo: siempre estamos hasta arriba, el seguir tal o cual metodología se asume como algo más a añadir a la lista de tareas. Somos, digamos, vagos a la hora de asumir y seguir disciplinadamente un método organizativo, cualquiera que sea éste.»

Lo siguiente no es más que una historia ficticia pero que repetimos continuamente; cualquiera que lleve sólo unos años en la profesión como desarrollador de software, se habrá encontrado con esta situación más veces de las que le gustaría.

Hubo una vez una ciudad que encargó la construcción de un rascacielos con el que deslumbrar a todo visitante: en él se alojarían importantes empresas e incluso parte de las dependencias del ayuntamiento. Además, tendría que ser muy alto y estético para que al cabo del tiempo formara parte del skyline de la ciudad.

La empresa constructora, nada más recibir el encargo, envió a las primeras cuadrillas de trabajadores a la zona para ir despejando la enorme parcela donde se construiría el emblemático edificio. Para ir ganando tiempo, los arquitectos de la compañía dejaron parcialmente los otros proyectos en los que estaban trabajando (…) para comenzar el que en adelante sería el proyecto estrella.

La licencias de obras se pidieron al mismo tiempo en que se taladraba la tierra para comenzar lo que serían los cimientos. Se reservaron por varios meses los cargamentos de hormigón a las compañías hormigoneras locales y antes de definir la estructura exterior, ya se estaba en conversaciones con cristalerías importantes a nivel nacional e internacional.

Al cabo de pocos meses los atónitos ciudadanos de la ciudad pudieron ver cómo surgía casi de la nada una estructura de hormigón que crecía día a día. Al medio año, las fachadas exteriores se cubrían con distintos diseños exteriores.

La primera crisis en el proyecto surgió cuando los aparejadores se dieron cuenta de que las acometidas de electricidad se habían dejado en el lugar equivocado (…), las de agua aún no se habían contratado con suficiente antelación y el material aislante no encajaba con la estructura dejada por el

último arquitecto que trabajó en el proyecto. Se resolvió frenéticamente rompiendo por aquí y por allá y con alguna comida en un lujoso restaurante con el gerente de la compañía de suministro de agua…

Al año, cuando ya se contaban las semanas para la inauguración y cuando ya estaban movilizados los medios informativos locales, un domingo plácido de verano, un ruido ensordecedor hizo saltar todas las alarmas. Los vecinos, aterrados, se asomaron a sus balcones y los que andaban por las calles vieron una enorme nube de polvo que ascendía hacia el cielo.

El emblemático edificio se había derrumbado cual castillo de naipes, el trabajo de cientos de obreros, técnicos e ingenieros se vino abajo en cuestión de minutos: en esos momentos ya había quien pensaba en las cabezas que rodarían por el estropicio mientras la ciudad perdía su magnífico e icónico edificio.

Pues bien, esta simple y ficticia historia que no tiene ni pies ni cabeza describe sencillamente cómo afrontamos la construcción del software la mayoría de las veces, incluido lo que se hace en grandes e importantes compañías.

¿A alguien le parecería ni medio normal que un rascacielos se construyera de tal manera? Esta desastrosa forma de trabajar es la habitual por *freelancers*, pequeñas empresas y grandes compañías multinacionales con departamentos de desarrollo. ¿Te suena? Sólo en reducidas islas de claridad y motivadas «desde dentro» por los mismos desarrolladores, se comienza a construir el edificio de un software, digamos, con el orden correcto.

¿Hay alguna relación entre la construcción de un automóvil, un edificio, una máquina de envasar aceitunas (por poner un ejemplo) y un producto software? Pues sí, la hay, y mucha: y es que todos esos entes de construcción que antes de existir han sido ideados en el mundo abstracto de ciertas cabezas pensantes, son terriblemente complejos y para su construcción ha sido necesario establecer un guion claro en todas las fases de su desarrollo.

Hablamos por tanto de gestionar y de llevar a cabo trabajos que en su conjunto no son sencillos, son imbricados y difíciles de abordar por una única persona o equipo de trabajo.

Al igual que ocurre con la producción de una película (alguien escribe un guion, éste se contrata, se localizan escenarios y se buscan actores, se graban las escenas, luego se produce y por último se hacen los trabajos de realización y comercialización), cualquier cosa mínimamente complicada debe tener y contar con un guion claro y adecuado a la naturaleza del trabajo a realizar. Esto que parece elemental y obvio para «otras actividades» no lo vemos igual de claro cuando construimos un software relativamente complejo

y que queremos que viva en el mercado varios años.

El software no escapa de esta complejidad y, por tanto, necesita de su propia hoja de ruta a riesgo de acabar como el edificio de la historia en ausencia de la misma: esta hoja de ruta no es más que la metodología de desarrollo.

Es difícil percibir que la metodología no sea una cosa más que se añade a las actividades propias de desarrollo, sino que forma parte intrínseca del mismo: siempre estamos hasta arriba de trabajo, el seguir tal o cual metodología se asume como algo más a añadir a la lista de tareas. Somos, digamos, vagos y perezosos a la hora de asumir y seguir disciplinadamente un método organizativo, cualquiera que sea éste. Por lo general se prefiere desarrollar aquellas partes del sistema que más nos atraen y dejar para lo último las más aburridas, sin hablar de otro tipo de tareas como generar una correcta documentación, guía de despliegue, pruebas de integración de calidad, etc. En ocasiones he tenido la impresión de que muchos desarrolladores intentan pasar la mayor parte del tiempo «jugando» a lo que les gusta, ignorando o esquivando todas aquellas tareas que no les reportan ningún tipo de diversión. En el mundo real, todo es importante para que un proyecto concluya con éxito.

¿Por qué no percibimos esa necesidad del mismo modo que a nadie se le ocurre levantar un edificio si no tiene un proyecto firmado por un señor arquitecto? Por cierto, ¿por qué a los arquitectos de la construcción se les pone casi siempre un «don» por delante y a los ingenieros informáticos no?, dejaremos esas consideraciones para otro momento…

Bromas aparte, sufrimos de esa tentación constante de ignorar la metodología por la misma naturaleza intangible del software: mientras que en un edificio se ven claramente paredes, ventanas, fontanería, calidades del suelo y puertas, etc. un cliente que nos encarga un sistema apenas va a percibir una interfaz de usuario y, si acaso, va a notar su estabilidad o inestabilidad: lo que hay detrás para que todo eso funcione correctamente lo ignora completamente, aunque esté podrido y a punto de derrumbarse. Sólo en proyectos con mucho volumen de negocio y un cliente, digamos, avanzado, he visto la exigencia de evidencias de la implantación de procesos de calidad, lo que incluye el uso correcto de una metodología.

Por otra parte, no vemos claramente que ese edificio que se puede caer (o que se derrumbará al más mínimo cambio meses más tarde) a quien aplastará en su caída es a nosotros mismos o si tenemos suerte a otros compañeros, lo cual deja bastante que desear en cuanto a nuestra honestidad profesional. Un edificio software que se cae a pedazos es un problema para quien lo tiene que volver a levantar, pero también un enorme y tremendo problema para la compañía en la que trabajamos por incurrir en costes extras que pueden poner en peligro el balance y viabilidad de la misma. No hablamos de cosas que no tienen consecuencias, sino que las tienen y muchísima.

Cuando construimos un sistema software con los pies de barro estamos dejando de lado la «rentabilidad metodológica», que no es más que construir bien ahora, en el momento adecuado de poner los cimientos del sistema y en sus etapas iniciales, siguiendo un guion bien establecido para avanzar con orden, para que una vez terminada y entregada la solución, el sistema pueda ser reutilizado, mantenido y evolucionado, multiplicando los resultados, la satisfacción del cliente y la nuestra propia por trabajar en algo bien construido y sin fallas. Si hacemos las cosas bien, más adelante obtendremos esa tan preciada rentabilidad en forma de ausencia de errores, capacidad de evolucionar el sistema, mayores ingresos, un cliente más satisfecho, etc. La metodología, desde luego, no es lo único para que esto sea así, pero sí es imprescindible.

Un desarrollador de software profesional no discute ni pone en duda el aplicar o no una metodología: forma parte de su ADN, de igual manera que un farmacéutico no entrega un medicamento sin receta o un constructor no pone en pie una casa sin el proyecto de un arquitecto.

¿Por qué entonces dejamos de lado demasiado a menudo la disciplina de implantar una metodología durante todas las fases de un proyecto? Ésta nos indicará cuándo y cómo avanzar en el proyecto: es el guion que nos dice cómo vamos progresando en el sistema, cómo trabajamos en grupo y qué tareas hace cada uno. Sin metodología nos costará muchísimo más saber qué está funcionando bien y qué mal, no sabremos si nos desviamos del objetivo y fechas de entrega o si vamos al ritmo adecuado, ignoraremos la estabilidad de lo ya construido y no tendremos manera de minimizar el número de errores del sistema.

Sin guion no hay película y de igual modo sin metodología no construiremos un software de éxito y de calidad.

El responsable del grupo de trabajo, manager, coordinador o jefe siempre es el que debe exigir la presencia y seguimiento de una metodología: si no lo hace, está ignorando una de sus responsabilidades.

La razón fundamental por la que eludimos el seguir correctamente un método organizativo a la hora de desarrollar software es porque nos gusta mucho la libertad de hacer algo (programar) cuándo y cómo queramos. Esta es una visión ingenua y cándida de nuestro trabajo; sólo cuando hemos visto que el hacer las cosas ordenadamente, con rigor y sin ir dejando cabos suelos, es entonces realmente cuando apreciamos el valor de trabajar con una metodología. Vale que en proyectos personales, ejemplos, pruebas de concepto o prototipos apliquemos el *cowboy coding*[10] y que nos divirtamos con ello, pero en un proyecto profesional que nosotros o la compañía para la que trabajamos va a rentabilizar, no podemos eludir la implantación de una metodología, cualquier que sea ésta.

En ocasiones sobrevaloramos la implantación de cualquier tipo de estrategia organizativa a la hora de hacer software. Una metodología, por más que nos machaquen las modas o los rigores académicos cuando estudiamos algunas de las formales, no es más que una serie de normas sobre cómo progresar en el desarrollo de un software, qué responsabilidades tiene cada rol y qué tipo de artefactos software y evidencias se deben construir.

En este sentido tenemos la capacidad de adaptar cualquier metodología a las necesidades reales o estructura particular de un proyecto.

Si apoyamos el desarrollo de un producto software bajo el paraguas de una metodología, la apropiada para la naturaleza del proyecto, avanzaremos con pies de plomo y rentabilizaremos su aplicación en forma de código más mantenible (menos errores) y simple de evolucionar (tiempos de adaptación a nuevos requerimientos más cortos), entre otras muchas ventajas.

Una de las leyes del software que he podido experimentar por mí mismo es que el esfuerzo dedicado a implantar y seguir disciplinadamente una metodología redundará proporcionalmente en una mayor rentabilidad técnica en el software que desarrollamos, esto es, si nos esforzamos por aplicar la metodología elegida, cualquiera que sea esta, mejoraremos la calidad del

producto en el que trabajamos en forma de menos errores y mejor y más fácil mantenimiento.

Puntos clave

- Un proyecto profesional debe contar con algún tipo de metodología; es más, se puede definir una mezcla de algunas existentes siempre y cuando se adapte bien a la naturaleza del proyecto.
- Como desarrolladores profesionales, debemos exigir siempre la implantación y seguimiento de una metodología.
- El supervisar el seguimiento de una metodología es una de las responsabilidades del gestor del grupo de trabajo.
- Si avanzamos en el proyecto correctamente, con orden y claridad, acumularemos «rentabilidad metodológica» de la que nosotros, el proyecto, la compañía y los mismos clientes ser beneficiarán más adelante.

Todo está en la UI

«[…] Una interfaz de usuario de calidad vende y aparenta más que el mejor back-end. Sólo se puede valorar algo que realmente no se ve con aspectos sobre los que proyecta su funcionamiento: una interfaz de usuario ágil, simple de usar y con atractivo, rapidez en el funcionamiento, funcionalidad para aportar valor al cliente (no para hacer más complicado su trabajo) y un larguísimo etcétera.»

Aún conservo en el la cajonera de mi puesto de trabajo un CD con toda la información, documentos y código de un producto en el que trabajé durante mis dos primeros años nada más terminar mi etapa académica. Si no lo he tirado aún ha sido por pura nostalgia, aunque tengo que reconocer que en aquella época fue cuando aprendí realmente a programar: eran los tiempos de C++, COM, DCOM, ActiveX, etc., cuando te podías pasar días enteros detectando un *memory leak* que hacía que un servicio se cerrara por *overflow* a las pocas horas de estar funcionando. Tardabas horas en implementar cosas que hoy día forman parte básica de un framework de más alto nivel. Cuando creí que ya «sabía» programar, cayó en mis manos la primera edición de «*Code Complete*», de Steven C. McConnel. Fue como una especie de epifanía por la que comencé a intuir que existen diversas formas de «programar mejor» y que además la calidad del software era un tema muy poliédrico y difícil de valorar incluso por los mismos profesionales.

Ese único CD es capaz de albergar todo el trabajo de un grupo de desarrollo de software de una diez personas durante dos años, cientos de miles de líneas de código, miles de pruebas, documentos escritos en español y traducidos al inglés, miles de artefactos software que, como se suele decir,

caben en la palma de una mano.

He aquí uno de los problemas fundamentales con los que nos enfrentamos los desarrolladores de software: la naturaleza intangible del mismo hace que sea extremadamente difícil valorar el esfuerzo dedicado a concebir, diseñar e implementar una solución. ¿Y ahí está el resultado del trabajo de toda esa gente durante tanto tiempo?, podría preguntar alguien ajeno a nuestra profesión.

Otro tipo de productos proyectan de alguna manera el esfuerzo que se ha tenido que emplear para producirlos, como por ejemplo un vehículo. Nadie imagina a una única persona diseñándolo y construyéndolo. Es fácil para cualquiera pensar en el enorme equipo de trabajo que ha participado en su desarrollo, probablemente miles de empleados. Un coche, un electrodoméstico, un teléfono móvil, cualquier cosa física desprende por sí misma y en cierta medida el esfuerzo que tuvo que aplicarse para desarrollarla y fabricarla precisamente porque son objetos tangibles, se pueden tocar, de modo que intuimos la enorme complejidad de sus mecanismos: no así el software en el que como mucho un cliente puede ver una interfaz de usuario, ciertos informes, que unos procesos se ejecutan a velocidad suficiente y cosas por el estilo. Complejas, sí, pero no «desprenden» de ningún modo su naturaleza más o menos complicada.

El principal problema es que puesto que no se ve claramente el esfuerzo de nuestro trabajo, sólo los que entienden ligeramente la naturaleza del desarrollo de software pueden imaginar el esfuerzo o dedicación necesarios para producir una aplicación.

Podemos valorar el dinero necesario para generar el contenido del CD que comentaba al comienzo: seguramente su coste total fue superior al medio millón de euros; ¿el resultado?, un producto software que cabe en un único y simple CD de varios céntimos…, imposible de valorar en términos «materiales».

Esta cualidad, digamos, del software, nos afecta de muchas maneras y a menudo cometemos el error de no tenerla en cuenta cuando abordamos los proyectos: menospreciamos el hecho de que un cliente pueda valorar injustamente el esfuerzo y dedicación necesarios para desarrollar algo porque sólo llega a percibir la punta del iceberg. ¿Podemos hacer algo al respecto?

Así es, podemos y debemos; nos encontramos ante uno de los problemas típicos de nuestra profesión: no nos preocupamos demasiado de mostrar el valor y el esfuerzo de lo que producimos.

Por otro lado, un nuevo elemento viene a complicar aún más este asunto: estamos inmersos totalmente en un paradigma por el que debemos ofrecer al usuario aplicaciones sencillas de usar y que al mismo tiempo escondan la enorme complejidad subyacente que deben tener para su funcionamiento, de ahí el dicho de que los manuales de usuario están hechos para no leerlos… Podemos poner el caso del buscador Google: interfaz de usuario completamente minimalista, ¿alguien se imagina las entrañas terriblemente complejas que debe haber detrás en cuanto a los algoritmos de búsqueda aparte de la descomunal infraestructura hardware que debe permitir que todo funcione tan maravillosamente rápido en un despliegue mundial del buscador? Sin embargo, nuestra puerta de entrada al mismo es un cuadro de texto, un botón y bonitas imágenes que van cambiando de vez en cuando, y todo parece que funciona mágicamente… Todo esto está perfectamente descrito en el fantástico libro que recomiendo de Steve Krug y de título «*Don't make me think! A Common Sense Approach to Web Usability*» («¡No me hagas pensar! Un enfoque de sentido común a la usabilidad web»); aunque enfocado a la web, los principios elementales de simplicidad son los mismos para cualquier otro escenario.

El ofrecer la funcionalidad al usuario de la manera más sencilla posible es en sí un reto de diseño y usabilidad y, al mismo tiempo y de manera paradójica, hace que sea aún más difícil que un tercero pueda valorar justamente el esfuerzo necesario de todo lo que hay detrás para que todo funcione correctamente. Estamos en la era de la simplicidad, en muchos sentidos, lo cual es en mi opinión una noticia magnífica ya que esta está ligada profundamente a la productividad.

Ignoramos y pasamos por alto en muchas ocasiones que el factor principal por el que se va a valorar la calidad de nuestro trabajo va a ser por la interfaz de usuario que seamos capaces de desarrollar, lo único realmente visible que podemos mostrar al mundo como fachada de nuestra aplicación: lo único que «se ve» y por tanto lo único que «se percibe». Aún más difícil lo tenemos cuando en cierta medida pretendemos simplificarla al máximo para

que sea sencilla y fácil de usar. Este es un dilema al que nos enfrentamos continuamente. Es cierto que se debería valorar por encima de todo cuánto mejoramos y el valor que aporta nuestra solución al negocio del cliente, pero este con frecuencia se ofusca al respecto si lo que ve es feo, poco intuitivo y complicado de usar.

Al igual que la expresión «no sólo hay que ser bueno sino parecerlo», en desarrollo de software podemos decir que todo está en la UI (*user interface* o interfaz de usuario), o al menos casi todo. Desde la UI un usuario percibe la calidad de la infraestructura software que la soporta.

Es cierto que hay muchos otros factores por los que un usuario o cliente nos puede valorar positiva o negativamente, como un buen soporte, que seamos capaces de resolver problemas o incorporar nuevas características rápidamente, que la documentación (por si acaso alguien la lee) es de calidad, etc., pero todas estas cualidades han pasado previamente por el filtro universal de una magnífica interfaz de usuario.

Recientemente hemos decidido cambiar las librerías de gráficas o *charts* que usábamos en la plataforma de software en la que trabajo actualmente. La anterior, aunque funcionalmente aceptable, parecía un poco insulsa… Al cambiarla por la nueva e integrar alguna característica más de las que trae por defecto, podemos decir que ahora aparecen ciertas gráficas de manera mucho más visual (con la capacidad de hacer zoom de bloques de datos, descargar una imagen de la gráfica, etc.). Son funcionalidades que seguramente el cliente va a usar mínimamente; sin embargo, el impacto que hace en demos y presentaciones es, sencillamente, espectacular.

Con muy poco esfuerzo hemos conseguido multiplicar la apreciación positiva de nuestros potenciales clientes, siendo la complejidad del sistema exactamente la misma.

Recuerdo perfectamente justo el caso contrario en una ocasión durante una etapa laboral anterior. Se desarrolló un framework para la gestión de un tipo de dispositivos cuyo uso era bastante complejo. Ese desarrollo simplificaba el trabajo con los mismos de manera que era mucho más trivial su integración en aplicaciones de más alto nivel. Igualmente se hizo un gran esfuerzo por establecer una estructura de grupo y metodología suficientes para el escenario en que nos movíamos (todos trabajábamos al mismo tiempo

en varios proyectos…).

Se habían dedicado ya muchas horas al proyecto cuando se presentaron los primeros resultados a la gente de otro departamento en teoría encargada de comercializar el producto. Me consta que habíamos hecho un gran trabajo, en las condiciones que teníamos, que además las cosas funcionaban relativamente bien, no había demasiadas deudas técnicas y estaban bien establecidas las bases para que el desarrollo creciera y evolucionara.

Sin embargo, el primer comentario que escuché en la presentación fue «pues vaya mier…», lo que lógicamente nos creó una sensación profunda de frustración.

El paso del tiempo te ayuda a analizar ciertas situaciones de modo que cuando pienso en aquella presentación ahora me doy cuenta de que quien falló fuimos justamente nosotros: nos centramos demasiado en el *back-end*, nada trivial, por cierto, sin tener en cuenta que este nunca iba a ser valorado suficientemente ni por nuestros clientes internos (de la compañía si los hay) ni externos (los finales). No pusimos el énfasis necesario para crear una interfaz de usuario agradable: era demasiado plano, con un diseño espantoso y además difícil de usar.

La lección de aquello fue clara: podríamos haber desarrollado una maravillosa interfaz de usuario, con un magnífico diseño y efectos agradables y seguramente la impresión del sistema habría sido fantástica, aunque internamente todo estuviera podrido y mal hecho. En aquella ocasión, ese tonto incidente plantó la semilla de las dudas acerca del proyecto, por lo que estamos hablando de cosas que realmente pueden tener un impacto muy importante en nuestro trabajo.

No sólo debemos realizar un buen trabajo, con los cimientos de unas normas para avanzar en él con orden (metodología), adquirir como hábitos las actividades de simplificar el diseño y refactorizar continuamente y, cómo no, respaldarlo todo por pruebas automatizadas; además debemos mostrar lo maravilloso que es todo con una interfaz de usuario que le guste a alguien que no tiene ni idea (ni tiene por qué tenerla) de qué se cuece internamente.

Al usuario realmente no le interesa lo que hemos sufrido para simplificar el diseño, ni si usamos fiel y disciplinadamente una metodología ágil fantástica; tampoco le importa el número de pruebas que nos garantiza que

cualquier nuevo bug introducido va a ser detectado rápidamente. El cliente sólo va a considerar el «valor» que le está aportando la solución, el tiempo que ahorra en implementar con ella ciertos procesos, la mejora sustancial que le aportamos en su negocio. La herramienta que tiene para percibir todo ese valor es indudablemente la interfaz de usuario, al menos para la mayoría de aplicaciones. Si esta falla, la apreciación positiva de todo lo demás caerá en picado.

Esto viene a ser como una ley que se cumple en el desarrollo de software: una interfaz de usuario de calidad vende y aparenta más que el mejor *back-end*. Sólo se puede valorar algo que realmente no se ve con aspectos sobre los que proyecta su funcionamiento: una interfaz de usuario ágil, simple de usar y atractiva, rapidez en el funcionamiento, funcionalidad para aportar valor al cliente (no para hacer más complicado su trabajo) y un larguísimo etcétera.

Si no mostramos lo bueno que es nuestro trabajo con esas cualidades externas, cometemos un gran error que puede poner incluso en peligro la viabilidad del proyecto. Es más, sabiendo que esto es así, lo podemos usar como herramienta a nuestro favor precisamente en momentos de duda: en lugar de centrar inicialmente los esfuerzos en el *back-end* (todo aquello que se cuece por debajo) se puede priorizar estratégicamente aquello que sí se ve y que se puede valorar mejor.

Puntos clave

- Debemos dedicar mucho esfuerzo a una interfaz de usuario ágil, sencilla de usar y con muy buen diseño. En muchas ocasiones vamos a obtener mucha más rentabilidad por este aspecto que por detalles internos.
- Un cliente interno (de la misma compañía) o externo no va nunca a valorar lo bien que hemos resuelto algo que es realmente complejo.
- El valor de un producto no viene por su maravillosa arquitectura o lo saneado que esté su código, sino por el valor y utilidad que sea capaz de aportar al usuario; para ello, en la mayoría de ocasiones, la interfaz de usuario es el peaje a nuestra aplicación.
- Lamentablemente, en ocasiones, la medida de progreso para nuestro jefe o manager no es más que lo que «puede ver» a través de la interfaz de

usuario.

Diletantismo tecnológico

«[…] sólo tendremos éxito si sabemos emplear la tecnología, cualquiera que sea esta, para resolver un problema y aportarle valor a alguien o a un negocio, o lo que es lo mismo, nuestras posibilidades de tener éxito se reducen si sólo nos ocupamos en conocer más y más tecnologías sin profundizar en ninguna y ¡sin resolver nada ni hacer nada de utilidad!»

A menudo se dice que todo lo relacionado con Internet y las Tecnologías de la Información va demasiado rápido: en poco tiempo alguien puede pasar de estar al día a sentirse completamente desfasado y obsoleto. Es cierto que las cosas se mueven a velocidad vertiginosa, y no estamos hablando del flujo enorme de información que nos llega continuamente, sino de las formas, usos y nuevas herramientas que aparecen compitiendo unas con otras. En este sentido, cualquier usuario que utiliza habitualmente un ordenador puede sentirse abrumado si intenta seguir este ritmo frenético. Parece como si en este ámbito el «estar al día» no es más que el conocer (aunque sea por encima) las últimas novedades en herramientas, aplicaciones, nuevos dispositivos, etc., aunque el uso que se haga de ellas sea el mismo que hace diez años.

Este fenómeno, estresante en mi opinión, nos afecta también a los desarrolladores de software; digamos que no sólo somos usuarios avanzados de herramientas, aplicaciones y utilidades para el trabajo o el ocio de alcance popular y doméstico, además, sufrimos el mismo tipo de avalancha sobre los tipos tecnologías que se usan para el desarrollo de esas mismas herramientas, aplicaciones y utilidades. Digamos que en este sentido somos bombardeados

doblemente por tierra, mar y aire.

En los años que llevo desarrollando software, unos quince en el momento de escribir esto, he visto un poco de todo; pero es que todavía ni intuyo lo que me queda por ver:

- En este tiempo he usado en profundidad cinco versiones distintas del entorno IDE con el que he estado programando un 80% de mi tiempo.
- He tenido que utilizar tres tipos de sistemas de control de versiones (estoy hablando de Visual Source Safe, Subversion y el repositorio de código del Team Foundation Server). Ahora me estoy familiarizando con uno que me gusta especialmente (GIT).
- He pasado de usar una metodología más o menos formal, a otro periodo en el que la palabra metodología no existía hasta abrazar completamente el enfoque ágil del desarrollo de software.
- He trabajado con varias tecnologías distintas para las interfaces de usuario (MFC, ActiveX, ASP, ASP.NET, ASP.NET MVC, *templates* con PHP, jQuery, javascript, JFC/swing, HTML5, etc.).
- He visto cómo ha evolucionado enormemente el lenguaje de programación que más he usado en los últimos cinco años (C#).
- He usado varias versiones distintas de servidor de base de datos del mismo fabricante; en cuanto a las bases de datos, ahora comienzo a poner en práctica algunas pruebas sobre bases de datos no relacionales enfocadas al *big data*.

Y un larguísimo etcétera que me hace llegar a la siguiente conclusión: seguramente las herramientas, entornos y formas de trabajo actuales de un desarrollador de software no tengan nada que ver con las que va a tener que usar de aquí a cinco años. Es como si de manera obligada tuviéramos que reinventar nuestra profesión cada cierto tiempo.

Un desarrollador de software que se inicie en la profesión tiene que tener esto muy claro; no existen zonas de confort que duren más de un año; si te empeñas en el inmovilismo y en no salir de aquello que conoces, estarás fuera del mercado antes que tarde.

Esto parece evidente, pero no lo es tanto el esfuerzo tremendo en horas de

dedicación, adaptación, nuevo aprendizaje, etc. que se emplea cuando se salta de una tecnología bien conocida a otra. Pocas profesiones tienen un ritmo de adaptación y evolución tan alto como el desarrollo de software. Ignoramos el coste profesional que esto tiene en nuestra carrera laboral.

Este cambio tras cambio de manera vertiginosa es causa de estrés en quienes descubren a los pocos años de desembarcar en el mundo de la empresa que esta característica es intrínseca y forma parte de nuestra profesión: o te adaptas o mueres (quiero decir, te quedas en la mayor obsolescencia, perdiendo seguramente oportunidades económicas o laborales).

También es causa de querer escapar a esta dinámica de continua evolución y adaptación intentando acaparar puestos de gestión, pero ese es otro tema… Me pregunto si esta es una de las razones por las que el número de desarrolladores de software va disminuyendo a partir de cierta edad. Es raro encontrar programadores profesionales de más de 45 años, lo que es una auténtica tragedia para el talento cuando precisamente la experiencia y el *background* son muy valiosos para realizar buenos productos software.

Sufrimos una evolución frenética (en ocasiones por motivos comerciales de los grandes de Internet) en las tecnologías y herramientas que usamos, eso es innegable, pero la cuestión es que en muchos ámbitos «seguimos resolviendo los mismos problemas». ¿Y entonces?

He aquí el quid de la cuestión y el error fundamental que cometen en mi opinión quienes yo mismo llamo «diletantes tecnológicos»: el desarrollo de software profesional se confunde a veces con ese maratón para estar «al tanto» de lo último de lo último, sin haber profundizado apenas en lo que tenemos en ese momento entre manos.

Olvidamos que somos programadores profesionales única y exclusivamente para «resolver problemas y dar utilidad» a otros. Las tecnologías que usamos para ello no son un fin en sí mismo, sino el medio para aportar valor y crear algo útil, lo que me trae a la mente uno de los principios del desarrollo ágil «*Working software is the primary measure of progress*» (la principal medida de progreso es el software que funciona).

Sin caer en la simpleza de criticar a los demás, veo cómo se dedican (y desperdician) ingentes cantidades de tiempo en leer blogs de tecnología y

gadgets, picotear aquí y allí sin profundizar en nada, mientras se intenta salir del paso de la manera más rápida (y menos profesional) en aquello que nuestro trabajo actual nos demanda.

¿Por qué entonces nos gusta ese diluvio de noticias, actualizaciones y nuevos modos de trabajo cuando ya sabemos que la mayoría de todo ello quedará obsoleto en tan poco tiempo? Sencillamente porque no reparamos lo suficiente en el esfuerzo que dedicamos a ese tipo de cotilleo vanidoso en lugar de crear soluciones, innovar con ideas emprendedoras y solucionar problemas reales.

Hace algún tiempo, cuando en una conversación de cafetería escuchaba a unos amigos (mucho más jóvenes que yo, todo hay que decirlo) comentar con entusiasmo las cualidades maravillosas de cierto framework, pensaba para mis adentros algo como «pobrecillos, la vida, Bill Gates o quien sea aún no les ha enseñado que en un par de años no habrá quien se acuerde de todo ese batiburrillo que me están contando, brbrbrbrbrbrbrbrbrbrbr».

No quiero con este discurso crítico dar la idea de que nos tenemos que alejar de ese ritmo frenético de novedades, ni mucho menos, sino resaltar que perdemos gran parte de nuestro esfuerzo en intentar siquiera seguirlo un poco. En cierta medida, los desarrolladores de software profesionales, tenemos y debemos estar al tanto de la última ola (el *state-of-the-art* tecnológico, que así parece que suena algo mejor), pero sin olvidarnos de una cosa: nuestro esfuerzo se debe centrar en resolver problemas.

He aquí una ley del software que me gusta repetir a modo de mantra: sólo tendremos éxito si sabemos emplear la tecnología, cualquiera que sea esta, para resolver un problema y aportarle valor a alguien o a un negocio, o lo que es lo mismo, nuestras posibilidades de tener éxito se reducen si «sólo» nos ocupamos en conocer más y más tecnologías sin profundizar en ninguna y ¡sin resolver nada ni hacer nada de utilidad!

Lo primero es el problema, después se elige cómo y con qué tecnología solucionarlo, no al revés. Lo que nos da prestigio como profesionales son los problemas que resolvemos, después y en segundo lugar el cómo los hemos resuelto.

Como siempre es el sentido común el que al final nos acerca a lo correcto; como informáticos y desarrolladores, únicamente vamos a tener

capacidad para profundizar en muy pocas tecnologías, sí para conocer muy por encima muchas otras.

Si es verdad que nos estamos acercando a un nuevo paradigma laboral según el cual sólo valdrá aquello de valor que podemos aportar a una organización y los resultados, no tiene ningún sentido que llenemos nuestro currículum vitae o nuestro perfil de LinkedIn con cosas como «Programador experto javascript, prototype y jQuery desde hace tres años. Experto en ASP.NET MVC. Conocimiento medio de Ruby on Rails. Alta experiencia en bases de datos Sql Server y MySql».

O bien algo como, «.NET Framework 4, WF, WCF, Entity Framework, Sql Server 2008, IIS 7, AppFabric, Team Foundation System, Visual Studio 2010, C#, ASP.NET, Visual Basic, Active Directory, SCRUM, GWT, PaaS: Google App Engine» (lo siento, he copiado y pegado de un perfil de LinkedIn a modo de ejemplo…).

No está mal resaltar qué tecnologías conocemos, pero tenemos que tener en cuenta que eso no dice «qué podemos» hacer con ellas.

Este es el producto de lo que yo llamo «diletantes tecnológicos», programadores que seguro que son muy buenos profesionales, apasionados de la tecnología, con gran talento y que además tienen la suerte de disfrutar conociendo nuevos caminos e indagando por aquí y por allá. No obstante, me temo que comenzamos a vivir en un mundo globalizado en el que cada vez es más fácil contratar a golpe de clic a un ingeniero indio muy bien preparado y que domina esas mismas tecnologías igual que tú; entonces, ¿qué nos puede diferenciar competitivamente? Únicamente los problemas que somos capaces de resolver.

Un desarrollador de software profesional sólo puede hablar de sí mismo exponiendo los trabajos que ha realizado (los problemas que ha resuelto), independientemente de las tecnologías que haya usado; en definitiva nadie nos va a examinar realmente del nivel que tenemos en todas y cada una de aquellos temas que indicamos en nuestro currículum que conocemos, lo que hace que tenga el mismo valor que el papel mojado.

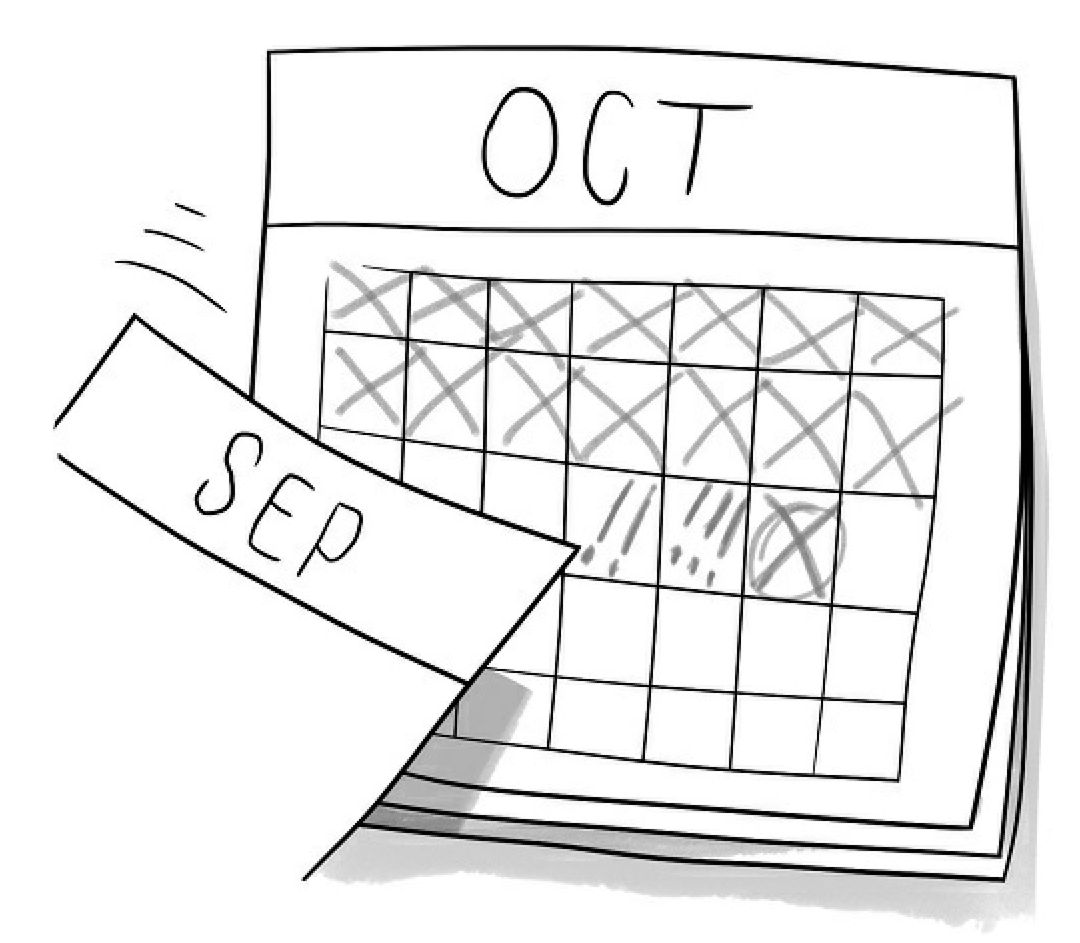
Si somos profesionales en tanto que resolvemos problemas, entonces debemos exponer nuestras habilidades del modo «Co-fundador de una web social para la valoración de vinos de calidad (implementada en X)»,

«Desarrollo con tecnologías de Microsoft de una aplicación para la monitorización de servidores», «Diseño y desarrollo de un sistema para la gestión de *smart meters* actualmente en producción de X compañías de electricidad», «Gestor de grupos de trabajo con *scrum*, Team Foundation Server desde hace tres años», «Autor de la guía de usuario para X», etc. Resultados, resultados y resultados.

Esos son los problemas que resolvemos realmente en nuestro día a día y que llevan nuestra firma, lo único que podemos exponer acerca de la calidad de nuestro trabajo, nada peor para el desarrollo de una carrera profesional como programador experto que caer en un diletantismo tecnológico exagerado y por sí mismo.

Puntos clave

- Se nos va a valorar más profesionalmente por los trabajos que hemos hecho (los problemas que hemos resuelto) que por decir que conocemos esa lista interminable de tecnologías, frameworks y formas de hacer nudos… Esto es una versión tecnológica de aquello de que «somos lo que hacemos, no lo que decimos».
- Es cierto que tenemos y debemos estar al tanto de lo que se mueve (a veces somos nosotros los mismos que provocamos ese movimiento), pero esto sólo nos debe consumir una parte pequeña de nuestro esfuerzo y tiempo productivos.
- El diletantismo tecnológico tiene más que ver con una vanidad ingenua que con querer conocer tecnologías para innovar y resolver problemas.
- Es frustrante saber que muchas de las tecnologías que usamos ahora mismo padecen una obsolescencia programada, pero esto forma parte de nuestra profesión y si esta idea no la digerimos bien, estaremos fuera del mercado antes de tiempo.



UN BUEN TRABAJO SIEMPRE NACE DE LA BUENA ORGANIZACIÓN

Este es uno de los secretos mejor guardados: sólo podemos conseguir buenos y abundantes resultados si nos organizamos adecuadamente; para ello, debemos dedicar algo de nuestro tiempo a planificar las tareas que tenemos en las próximas semanas y comprometernos con los objetivos que nos marcamos.

No se trata de trabajar más horas, sino de trabajar mejor

«A menudo se confunde la productividad con trabajar más y más horas. La productividad es la capacidad de trabajar mejor dedicando menos recursos para obtener un mejor resultado, lo opuesto del síndrome del calienta sillas, la falta de organización, entornos de trabajo agresivos, interrupciones continuas o la inexistencia de una simple planificación. La calidad de lo que producimos está relacionada con la buena productividad. Esto es así en cualquier trabajo, también en el desarrollo de software.»

Hay ciertas realidades que abruman por su magnífica sencillez: en software, trabajar más horas no es aumentar la productividad, que a nadie le sorprenda que no por echar más tiempo en nuestro trabajo se va a producir más y con la calidad óptima esperada.

Lamentablemente todos hemos «calentado silla» en algún momento durante la realización de un proyecto y esto no está mal siempre y cuando sea algo puntual; forma parte de la cultura del presentismo laboral. Ahora bien, cuando se necesita echar más horas de las habituales como algo crónico no hay duda de que algo no funciona en la organización, la forma de abordar los proyectos o hemos asumido mucho más trabajo del que podemos abarcar, por poner unos ejemplos.

La creación de software, como actividad con una gran parte de naturaleza creativa, sólo puede surgir cuando se dan las condiciones adecuadas para tomar en cada momento las decisiones correctas. Apenas nos damos cuenta de que cuando programamos, tomamos decisiones de diseño continuamente.

Difícilmente vamos a acertar cuando tenemos diez horas a la espalda de cansancio, imposible realizar el esfuerzo de refactorizar aquello que sabes que podría quedar mejor cuando no ves la hora de salir de la oficina. Así, poco a poco, en este tipo de condiciones es cuando se van acumulando las deudas técnicas que hacen que al final la calidad del producto o proyecto realizado baje varios escalones. No me puedo cansar de repetirlo: somos profesionales en tanto la calidad de lo que producimos es buena o excelente. Podemos competir produciendo en cantidad y baja calidad, pero esto, en mi opinión, tiene poco recorrido a medio plazo.

Lamentablemente existe mucha mentalidad de gestor según la cual si un empleado fabrica veinte tornillos a la hora, cuarenta a las dos horas y así sucesivamente, entonces un desarrollador de software va a ¡producir más si trabaja más tiempo! Nada más lejos de la realidad. Espero que se me entienda: un programador no es una raza especial de empleado, ni mucho menos; sí es verdad que nuestra atención plena a los detalles y la creatividad en el diseño sólo la podemos mantener unas pocas horas al día, no hablemos ya de hacer una retrospectiva de lo realizado hasta el momento para afrontar mejor las futuras fases de desarrollo; esto afecta y mucho a la calidad de todos los artefactos software que producimos.

¿Y qué más da que lo entregado tenga peor calidad «interna» si al cliente le funciona? Espero que a estas alturas de *El Libro Negro del Programador* esta pregunta se pueda responder fácilmente. Esa mala calidad interna provocará que el software sea difícil y caro de mantener y difícil y caro de evolucionar (rentabilizarlo aún más en el futuro). Esta idea cuesta muchísimo hacerla entender a quienes no «viven» el software desde dentro. Es cierto que el desarrollo ágil indica que la única medida de progreso de un software es que funcione o no (*working software*), aunque hay que matizar que si no está hecho con la suficiente sencillez de diseño, limpieza de código, buenas prácticas de desarrollo, etc. lo que hagamos en fases posteriores será caro de producir o imposible de realizar.

¿Cómo llegamos a la situación en la que no tenemos más remedio que echar más y más horas cada día en la oficina? Tanto si trabajas en una compañía «a sueldo» como si eres *freelance* puedes sufrir igualmente las causas que hacen que trabajes de una manera totalmente improductiva.

Comenzar un proyecto software con recursos y tiempo suficientes es uno de esos momentos más gratificantes de nuestra profesión: las ideas están claras, sabemos lo que hay que hacer, tenemos ahí un montón de cosas en las que investigar, la oportunidad de participar en *dojos* y de hacer *spikes* de conceptos antes de plantear mínimamente la estructura del sistema, buscar frameworks en los que apoyarnos, podemos incluso dedicar tiempo a evaluar las herramientas a usar y, en ocasiones, ¡hasta la metodología!, y todo, porque tenemos tiempo suficiente por delante (o creemos que lo tenemos). Si no tenemos encima un manager o gestor de proyecto consciente de esto y que controle y planifique los tiempos correctamente, lo habitual es que comencemos los proyectos relajadamente.

Ese es el error fundamental en el que solemos caer todos ante un proyecto incipiente. Me temo que te das cuenta de esta dinámica cuando has cometido ya este error y cuentas con algunos años de experiencia.

A medida que avanza el tiempo vamos comprobando cómo hay una sensación creciente de que no vamos a llegar a las fechas previstas y, al llegar a ese punto es cuando el barco comienza a hacer aguas relajando la metodología, las pruebas y, en definitiva, la calidad de lo que se desarrolla. Cuando hablamos de «calidad» nos referimos también al coste y la rentabilidad que se va a poder obtener por ese software.

Quizá por un efecto de responsabilidad o porque el responsable comienza a apretarnos las tuercas, la jornada laboral se va a alargando dramáticamente. Si tenemos en cuenta que desarrollar software es una actividad creativa, similar a la artística que realiza un escultor o un pintor abstraído en su propia obra, entonces podremos ver fácilmente que ese estado de creatividad no puede durar ocho, nueve, diez horas al día. Es más: sostengo que no puede durar ni cinco.

El desenlace de la historia es que semanas antes de la entrega es cuando se echan más horas que nunca, pero no son horas completamente productivas, sino que se trabaja a salto de mata con el objetivo de entregar «lo que sea» que nos permita salir del paso; cuando nuestra mente está invadida por ese objetivo, los de la calidad, refactorings, buena cobertura de código, etc. no tiene cabida. Así de simple, así de sencillo.

Por tanto, aunque lleguemos a los plazos comprometidos, habremos

hecho las cosas tan mal que nunca mejor aplicar lo de «pan para hoy y hambre para mañana»: el precio que pagamos entonces es el de alargar todavía más las fases posteriores del proyecto (por la inmantenibilidad del software y multitud de errores no detectados por no seguir correctamente la metodología, si es que ésta no se terminó abandonando). El problema entonces tiene difícil solución.

Vemos cómo partiendo de unas condiciones ideales, terminamos echando horas y quemando asiento. Cuando eso ocurre y vemos a la gente con los ojos encendidos a las ocho de la noche, no podemos decir que estén trabajando productivamente: esa hora indica precisamente la falta de productividad durante las semanas o meses anteriores.

Quizá a algún jefe le guste ver a sus empleados tan «comprometidos» con los objetivos de la compañía al quedarse continuamente más y más horas en la oficina. Esto puede impresionar y mejorar tu valoración entre «cierto tipo de jefes», pero la realidad es muy distinta: el hecho de necesitar más tiempo de la jornada laboral es un fracaso en toda regla, un problema que hay que resolver y un síntoma claro de que algo no va bien en esa organización.

Siempre me he preguntado por qué en muchas de las ocasiones en las que he participado en un proyecto se nos ha echado el tiempo encima. Ahora soy muy consciente de la respuesta y de los malos hábitos laborales y organizativos que provocan esta situación. Y es que no debemos olvidar que trabajar periodos prolongados con estrés y exceso de horas provoca un resultado de mala calidad y esto es lo que siempre intenta evitar un buen profesional.

De lo que estamos hablando no es más que de saber trabajar productivamente, lo cual es lo opuesto a resolver una crisis de tiempo dedicando más horas a un trabajo que vamos a hacer mucho peor en esas condiciones.

Son muchos los factores que provocan que se llegue a este tipo de situaciones. A continuación analizamos los más comunes que yo mismo he sufrido personalmente en algún momento.

» No sabemos qué tenemos que hacer con la suficiente antelación

Recuerdo hace unos años que me sorprendí a mí mismo al darme cuenta un

domingo por la noche de que no sabía exactamente qué tenía que hacer en la oficina el lunes por la mañana. Sólo tenía que llegar y según la avalancha de correos atender unas cosas u otras según la importancia, las ganas que tuviera en ese momento o la prioridad que «otros» le dieran a tal o cual asunto. Trabajo tenía, y mucho, pero con ausencia total de fechas claras para su finalización. El resultado de esta situación es que como por arte de birlibirloque se presentaba una crisis un día porque «alguien» demandaba algo para el día siguiente. Se hacía software, sí, pero con una organización que dejaba mucho que desear y, por tanto, la calidad de lo que desarrollábamos no era para tirar cohetes, me temo.

» Continuamente nos llegan tareas no previstas

Este es uno de los síntomas claros de falta de previsión y planificación. Es inevitable en cierta medida que surjan asuntos importantes sin que nosotros o nuestro manager pueda remediarlo: una llamada urgente de un cliente, una oferta importante que hay que atender, etc. Ahora bien, cuando las tareas no previstas surgen continuamente como algo habitual en el modus operandi de la compañía, estamos ante un grave problema que provocará que seamos incapaces de planificar nada ni de hacer software con la suficiente dedicación. La carrera hacia el exceso de horas está servida.

» Las deudas técnicas nos pasan factura a menudo

Esto ocurre cuando detectamos con frecuencia que por no haber mejorado algo en su momento, un refactoring crítico que no hicimos, ahora nos cuesta más avanzar en el proyecto.

Las deudas técnicas en un software dan la cara tarde o temprano, precisamente por eso las llamamos «deudas», porque son algo que de no resolverse, nos va a costar más tiempo y esfuerzo resolverlas más adelante o van a provocar efectos secundarios desagradables.

El día a día tiene muchos momentos para obtener pequeñas satisfacciones: saber que se ha resuelto un problema o se ha realizado un microdiseño de manera genial, a mí al menos, me aporta un pequeño rayo de alegría y satisfacción por el trabajo realizado. Convertimos esa pieza bien terminada en una joya que más adelante nos va a reportar facilidad y

comodidad a la hora de resolver problemas.

» La planificación de las fases del proyecto no es estable
A veces pienso si es que yo he tenido demasiada mala suerte, pero esto no es más que una percepción personal: cuando hablo con gente con experiencia más o menos dilatada, llego a la conclusión de que los mismos problemas suceden en muchos entornos de trabajo distintos. A menudo ocurre que en medio de las tareas de una planificación, el gestor del proyecto decide «por su cuenta y riesgo» cambiarla. Puede que haya buenas razones para ello pero si esto ocurre a menudo es síntoma de que el gestor del equipo, el responsable de la planificación de las tareas, no está realizando bien su trabajo. En software nos organizamos con las tareas que tenemos por delante en mente: si estas cambian continuamente padeceremos una incertidumbre que afectará negativamente al trabajo realizado. Ahora mismo soy responsable precisamente de definir y planificar estas fases de trabajo e intento muy cuidadosamente no salirnos del guion establecido; no siempre lo consigo, pero sé claramente que es mi responsabilidad que no nos salgamos durante ese tiempo de la planificación establecida.

» No tenemos un ambiente de trabajo lo suficientemente tranquilo para poder fluir mientras programamos
No es que necesitemos un convento donde poder trabajar en paz, pero sí es necesario un ambiente que nos permita trabajar con la suficiente tranquilidad y serenidad. No hace mucho leí que en un *open space* (esos espacios abiertos de oficinas en donde nos sentamos en largas mesas y podemos ver las caras de casi todo el mundo) un trabajador recibía entre cincuenta y cien «motivos de distracción» durante su jornada laboral. Vale, de acuerdo, en Internet podemos encontrar para cualquier tema una cosa y la opuesta, pero es que esto mismo lo he vivido personalmente.

En ocasiones cuando programamos estamos ensimismados en lo que hacemos, lo mismo que un escritor desarrollando un nuevo capítulo para su novela: esa actividad le exige una atención y concentración plenas; ¿cómo la vamos a obtener si padecemos continuamente distracciones? Alguien puede pensar que qué gente «más delicada», pues lo siento mucho, tanto si estás

haciendo una escultura, escribiendo una novela o pintando con chocolate el dibujo de Mickey en una tarta de cumpleaños, necesitarás al menos un periodo de concentración más o menos prolongado. Negarlo es engañarnos a nosotros mismos. No podemos crear si nuestra mente no puede permanecer concentrada. Todavía no me creo eso de poder pensar en dos cosas a la vez.

» No dominamos correctamente la tecnología que estamos empleando
Esto es un gran lastre para poder trabajar con productividad: si no conocemos con la suficiente profundidad la tecnología que usamos no podemos esperar realizar grandes progresos en el software que tenemos que realizar. Es nuestra responsabilidad detectar esta laguna y formarnos o pedir formación a la compañía para poder avanzar mejor en nuestro trabajo. Si esto es así, el responsable del equipo debería delegar las tareas más críticas a los miembros con más experiencia y dejar las menos relevantes a quienes aún les falta esa mayor madurez en las herramientas o tecnologías a usar, al menos mientras dura ese periodo de formación. Vemos de nuevo cómo una decisión de gestión puede lastrar la productividad de un equipo.

» No tenemos la suficiente capacidad para realizar con orden las tareas pendientes
Trabajar con orden es fundamental. No es raro ver a quienes van saltando de una tarea a otra sin terminarlas completamente. No hablemos ya de que comenzamos siempre por aquellas que más nos atraen.

Como programadores, el saltar de una tarea otra, al igual que concentrarnos en un tema y luego en otro diferente, conlleva un tiempo. Se dice que cuando perdemos la concentración en algo por una interrupción, tardamos quince minutos al menos en volver al mismo nivel de concentración. Si flirteamos con varias tareas al mismo tiempo vamos a necesitar mucho más esfuerzo y tiempo en completarlas que si las vamos realizando una tras otra. Sobre cómo organizarnos bien, destacan los libros de David Allen al respecto.

Lo siento, no somos seres con pensamientos paralelos, sólo podemos estar muy concentrados en un tema a la vez.

» No siempre tenemos los equipos, herramientas y el entorno adecuados

Ya he comentado en algún capítulo aquella experiencia traumática en la que compilar el proyecto completo en el que tenía que trabajar tardaba más de cinco minutos en un PC «de la época». Debemos exigir lo que necesitamos para trabajar con comodidad y productivamente. En ocasiones es responsabilidad nuestra detectar estas necesidades porque puede que el responsable del equipo no sea consciente de ellas.

» Nos acompaña en el equipo algún vampiro de tiempo
Debemos cortar de raíz a aquellos que sencillamente no nos dejan trabajar provocando continuas interrupciones, peticiones de ayuda reiteradas, correos estúpidos, etc. Son los que yo llamo «vampiros de tiempo». Como todo, el sentido común nos indica dónde está la barrera entre lo admisible y el abuso. En esto debemos ser drásticos: un compañero así, por bien que nos caiga, nos está chupando la sangre, o sea, no nos está dejando ser lo profesionales que debemos ser. Debemos intentar ser nosotros mismos quienes gestionamos nuestro propio tiempo y que éste no esté a merced de los caprichos e intereses de los demás.

A menudo se confunde la productividad con trabajar más y más horas. La productividad es la capacidad de trabajar mejor dedicando menos recursos para obtener un mejor resultado, lo opuesto del síndrome del calienta sillas, la falta de organización, entornos de trabajo agresivos, interrupciones continuas o la inexistencia de una simple planificación. La calidad de lo que producimos está relacionada con la buena productividad. Esto es así en cualquier trabajo, también en el desarrollo de software.

Como vemos, hay multitud de factores que determinan si trabajamos productivamente o no. Algunos de esos factores tenemos la responsabilidad de controlarlos y gestionarlos nosotros mismos; otros, en cambio, son responsabilidad del gestor del equipo y de la organización.

Queda claro entonces que no producimos más trabajando muchas más horas, sino mejorando todos los aspectos relacionados con la productividad que rodean al trabajo que realizamos.

No programamos delante de un ordenador y ya está: hacen falta las condiciones adecuadas para hacer un buen trabajo; a veces lo que marca la

diferencia entre una compañía u otra, entre un proyecto y otro no es la calidad técnica del equipo, sino las condiciones que lo rodean.

Puntos clave

- Cuando es necesario y frecuente recurrir a las horas extras para terminar el trabajo, tenemos por delante un caso de falta de productividad en el equipo.
- No podemos mantener la calidad del trabajo después de un número de horas de dedicación intensa.
- Siempre buscamos hacer software de calidad y, para ello, debemos resolver los problemas de productividad, algunos de los cuales mencionamos en este capítulo.

Sobre frameworks, librerías y cómo reinventamos la rueda

«Frameworks, librerías o módulos son conceptos muy relacionados pero todos implican lo mismo: reutilizar, desacoplar funcionalidad de nuestros proyectos y centrarnos en los requisitos de alto nivel. En ocasiones reinventamos la rueda por la simple pereza de no investigar un poco lo que ya han realizado otros con éxito.»

«No hay ningún problema que no se pueda descomponer en otros más pequeños y abordables». Recuerdo como si fuese ayer esta afirmación de un profesor de Álgebra Matemática en mis primeros años de universidad. Esto puede parecer evidente pero alguien te lo tiene que señalar por primera vez para que te des cuenta de ello y todo lo que implica.

En cierto sentido, este principio lo podemos aplicar en el día a día en multitud de escenarios distintos. A veces nos sentimos agobiados por una nebulosa mental de color gris ante una inabarcable tarea que tenemos por delante y que sabemos que nos puede llevar días o semanas en resolver.

No obstante, si miramos bien y con perspicacia, podemos ir viendo cómo esa nebulosa amenazante tiene partes; a medida que miramos mejor esas partes van quedando cada vez mejor definidas y si auscultamos todavía más, llegamos a un punto en donde lo que antes era grande y casi irresoluble, ahora es un conjunto de pequeños problemas, problemillas, que individualmente son fáciles de resolver.

De repente nos damos cuenta de que lo que antes era enorme y no sabíamos cómo abordarlo, ahora ya tiene solución e incluso podemos anticiparnos al momento en que quede todo resuelto. Esto no es más que una

versión simplificada del «divide y vencerás».

Pues bien, el acto de programar y de resolver problemas mediante el software es muy parecido a este proceso, o al menos debería serlo.

La diferencia entre un buen programador y un desarrollador, digamos, principiante, está en que a este último le llega el problema enorme y se pone inmediatamente a tirar líneas de código. El primero se anticipa mucho mejor y, muchísimo antes siquiera de plantear el nombre de una clase o módulo, analiza el problema que tiene enfrente y lo divide en partes más pequeñas y simples; éstas en otras aún más pequeñas, así hasta que todas las piezas adquieren la forma de un mosaico coherente que nos da la suficiente confianza como para comenzar a codificar. Digamos que esto es un acto de diseño de muy alto nivel. Aún no tenemos en cuenta en este punto estrategias de refactoring y la aplicación de buenos principios: esta descomposición conceptual del problema es anterior.

Hace un tiempo me plantearon el desarrollo de un sencillo portal web a desarrollar en ASP.NET MVC en el que había que integrar, entre otras cosas, la inserción y obtención de imágenes de base de datos, autentificación por Gmail y Facebook (…), validación de formularios asíncronamente por ajax y un larguísimo etcétera. Pues bien, algunos de esos elementos los tenía meridianamente claros, otros eran un mundo totalmente desconocidos para mí en aquel momento.

Antes de ponerme manos a la obra y comenzar con el esqueleto general de la aplicación, investigué pormenorizadamente en proyectos de ejemplo todas y cada una de aquella funcionalidad que no tenía claro cómo afrontar. Cada uno de estos proyectos era pequeño pero resolvía un problema muy concreto que después podría integrar fácilmente (al conocer bien sus detalles internos) en el proyecto general. Con el tiempo, este iba creciendo ordenadamente, con cada uno de sus elementos funcionales bien resueltos, coherentes y sin cabos sueltos, sencillamente por haber hecho el ejercicio de subdividir en partes el problema general. Por el camino, descubrí multitud de librerías bastante populares que resolvían un gran número de esos problemas particulares.

Esto no es más que un simple ejemplo pero que podemos aplicar en cualquier momento en que tenemos algo que resolver de una mínima

complejidad.

Yo siempre digo que los desarrolladores de software en cierto sentido somos un tipo de traductores: dado un problema lo convertimos (traducimos) a un diseño que dará lugar a líneas de código que lo resuelven. Lo curioso de este proceso es que independiente de la tecnología, lenguajes o herramientas que usemos para resolver el problema, el «acercamiento mental» al asunto es idéntico.

El error fundamental que cometemos, insisto, es evitar este proceso de descomposición en partes más pequeñas y querer abordarlo todo de golpe porque para muchos lo más divertido es escribir código «de la manera que sea». Escribimos código, bien, pero necesitamos una guía que nos indique que el progreso y la manera de generar código productivo es el correcto, lo demás es pura improvisación con consecuencias desastrosas en la calidad de lo que generamos, la compañía para la que trabajamos o bien nuestra propia carrera profesional.

Es de vital importancia en nuestra profesión saber descomponer lo grande en partes más pequeñas, individuales, independientes y totalmente desacopladas, y aquí es donde llegamos realmente al quid de la cuestión: la incapacidad habitual de desarrollar software con un grado de desacoplamiento alto. Tanto es así que podemos afirmar que la calidad del software es directamente proporcional al nivel de desacoplamiento alcanzado por cada una de sus partes. Me sorprende que el concepto de «acoplamiento» no esté entre en boca continuamente de los desarrolladores de software; quizá nos gusta más jugar con tecnologías emergentes que en dominar bien las herramientas fundamentales para la realización de código correctamente (o, al menos, de la mejor manera posible según las circunstancias de cada uno).

Imaginemos por un momento que nos encargan desarrollar para un cliente un CRM (*customer relationship management*) con unas particularidades especiales. El cliente lo que quiere es un sistema «que le sirva», que le aporte valor y le da igual qué tenemos que usar y cómo lo usamos para conseguir este objetivo.

Con un sencillo análisis vemos que como mínimo el producto que le vamos a entregar tienen que tener una interfaz de usuario, un modelo de datos suficientemente rico y una lógica de negocio que conecte las interacciones

del usuario con la información que gestiona y almacena.

Ahora bien, «una interfaz de usuario» sigue siendo un concepto muy general, bajamos de nivel y según las especificaciones del cliente iremos obteniendo partes esenciales del sistema, como por ejemplo gestión de clientes, de proveedores, de contactos, conectarlo con la gestión del stock de la compañía y un larguísimo etcétera. Volviendo sobre el mismo proceso, vemos que la gestión de clientes implica el poder dar de alta nuevos, modificar los existentes, distinguir entre tipos de clientes, etc. Al poco tiempo que hagamos este ejercicio, habremos descompuesto un problema aparentemente grande (el desarrollo de un CRM) en partes mucho más pequeñas.

El cómo gestionemos, diseñemos e implementemos esas «partes» determinará el éxito del sistema, ni más ni menos; me temo que no siempre se ve esta relación que reconozco es un poco sutil.

La tentación habitual, sobre todo entre desarrolladores neófitos, es no llegar con la suficiente granularidad a la definición de esas partes, de modo que cuando llega el momento de codificar, el nivel de acoplamiento entre ellas es aún demasiado grande, si es que este trabajo de descomposición no lo ha hecho alguien de mayor experiencia.

Así es como empezamos a dejar piedras en al camino en forma de código y módulos «muy parecidos» o duplicados, por poner un ejemplo. Si se sigue así, se llegará a un sistema que puede que funcione, pero que cuando sufra la más mínima modificación en una isla de su estructura, saltará por los aires otra en cualquier parte del sistema de manera inesperada. Se habrá construido entonces algo totalmente rígido por y para ese cliente en donde se habrá desarrollado absolutamente todo (menos el motor de la base de datos, esperemos). Este es un extremo que, me temo, sucede a menudo: la creación de una solución ad-hoc para un cliente en donde nada, absolutamente nada es reutilizable y peor aún, visto desde el otro extremo, tampoco se ha reutilizado nada de otros. Digo, esto es un extremo, pero ¿quién no ha visto este planteamiento con sus propios ojos en alguna medida?, es más, ¿quién no ha caído en este error alguna vez?

Una de las habilidades que debemos tener para desarrollar software es un gran sentido común, aunque a veces necesitemos un mentor (o una pila de

libros) para señalarnos en qué consiste ese sentido especial que nos pueda ayudar a desempeñar nuestro trabajo cada vez mejor y con mayor eficacia.

Somos eficaces en la medida que «reutilizamos» y escribimos código reutilizable. Esta es una de mis «frases-mantra» que repito continuamente.

Si hemos conseguido llegar a un conjunto de partes suficientemente pequeñas y modulares, desacopladas entre ellas y que aportan individualmente cierta funcionalidad, no hay que ser muy listo para encontrar ciertos patrones entre esas mismas funcionalidades y quizás desarrollos de anteriores proyectos. Hemos llegado así de manera natural al concepto de librería o biblioteca: por fin hemos llegado a la conclusión de que hay partes comunes en muchos proyectos que, si se aíslan adecuadamente sin particularidades específicas, podremos reutilizar una y otra vez, ahorrando así tiempo de desarrollo y haciendo descansar nuestra solución en partes ya muy probadas y maduras. Evitamos, en suma, inventar la rueda continuamente.

Un paso más en este concepto de «aislamiento de funcionalidad reutilizable» lo proporciona un framework de trabajo; en este caso se establece también un conjunto de prácticas para resolver un conjunto de funcionalidades con el mismo criterio. La mayoría de las veces entendemos, porque son conceptos imprescindibles de un desarrollador de software, qué son y para qué sirven las librerías y frameworks, sin darnos cuenta de que en el día a día debemos trabajar con la misma filosofía: aislar lo más desacopladamente posible las funcionalidades de nuestras aplicaciones.

Cuando hablamos de trabajar con productividad y eficiencia al desarrollar software también nos referimos a la necesidad de buscar frameworks lo suficientemente maduros sobre los que construir nuestros proyectos. Es más, ¿no es así como han evolucionado las ciencias de la computación?, abstrayendo cada vez más, nos encontramos lenguajes y herramientas cada vez de más alto nivel que nos permiten escribir aplicaciones más rápidamente. El objetivo es que el desarrollador trabaje principalmente centrado en los requisitos del proyecto, no en los detalles de infraestructura o de bajo nivel, en la «fontanería».

¿Quién no ha escrito alguna vez una sencilla librería para escribir mensajes de log? Sin embargo, son bien conocidas librerías y de uso abierto como *log4net* (*port* de log4j para .NET), *Common.Logging* y un larguísimo

etcétera.

Así sucede lo mismo con partes de la infraestructura de nuestro proyecto que podemos hacer descansar sobre librerías y frameworks bien conocidos y consolidados.

Frameworks, librerías o módulos son conceptos muy relacionados pero todos implican lo mismo: reutilizar, desacoplar funcionalidad de nuestros proyectos y centrarnos en los requisitos. En ocasiones reinventamos la rueda por la simple pereza de no investigar un poco lo que ya han realizado otros con éxito.

Chocamos a veces ante la «pereza del programador»: en ocasiones nos resulta más cómodo hacerlo uno mismo que buscar si a alguien se la ocurrido hacer lo mismo de manera general y elegante y publicar su trabajo. Si caemos en este tipo de pereza creedme cuando os digo que saldrá muy caro a medio plazo, pues entonces no sólo tendremos que preocuparnos de que la esencia y lo importante para el proyecto funcione, también tendremos que mantener todas esas partes que no hemos querido reutilizar y realizadas por otros seguramente mejor que por nosotros.

En un simple proyecto de la plataforma en la que estoy trabajando actualmente, puedo contar hasta ocho frameworks y librerías distintas que con infinita generosidad, sus autores han puesto a disposición de todo el mundo. Es impensable que en su día tuviéramos que desarrollar toda esa funcionalidad desde cero.

Puntos clave

- Si no sabemos afrontar una funcionalidad muy específica, mejor investigar sobre ella en un proyecto aparte, no en el general. De este modo, los experimentos no ensucian el código de producción.
- Nos gusta innovar, claro está, pero en la mayoría de las ocasiones muchas utilidades ya han sido creadas y puestas a disposición de la comunidad de desarrolladores con licencias abiertas que nos permiten reutilizarlas.
- Nuestro proyecto será mucho más mantenible si conseguimos desacoplar al máximo sus partes funcionales.

- No intentemos reinventar la rueda, esto sólo sirve para practicar y experimentar; en un proyecto serio debemos reutilizar al máximo librerías y frameworks bien conocidos, maduros y consolidados.

Los buenos desarrolladores escriben código depurable

«La calidad de una pieza de código es directamente proporcional a la capacidad que presente para su depuración.»

A veces cuando me preguntan a qué me dedico exactamente, me encuentro con algún problema por no poder sintetizar correcta y brevemente mi ocupación actual, de modo que suelo tirar por la calle de en medio, como se suele decir, contestando entonces que «hago software» o «programo».

El caso es que cuando me quedo pensando en esta respuesta en verdad me digo a mí mismo que no es del todo cierto, pues aunque me dedicara todo el tiempo a desarrollar software, a programar, ¿cuánta parte de ese tiempo está dedicada realmente a escribir «código de producción»? Entendemos como código de producción aquél que realmente resuelve los problemas de la aplicación, el que se ejecuta cuando se despliega para el cliente; no obstante, para llegar a un código de producción de buena calidad, hace falta mucho trabajo alrededor del mismo en forma de organización, disciplina, aplicación de buenas prácticas, etc.

Este es un asunto que confunden muchos desarrolladores: pasamos poco tiempo realmente escribiendo código nuevo de producción, el resto lo dedicamos a muchos otros factores que «rodean» esta actividad y que bien describimos a lo largo de *El Libro Negro del Programador*; no obstante, a muchos de esos otros factores no le damos la relevancia que tienen. Sostengo que gracias a esos factores, podemos desarrollar código final de producción de calidad.

Lo que más nos gusta, quizá por ser la parte más creativa del trabajo, es

sentarnos, resolver problemas escribiendo código, indagar tal o cual tecnología, etc., esto es lo que yo llamo la parte lúdica del desarrollo de software. No obstante, sería iluso y poco productivo intentar dedicarnos a esto todo el tiempo.

¿En qué se nos va entonces gran parte del tiempo? En cualquier actividad, para centrarte realmente en un asunto, necesitas algún tipo de organización y planificación. En software, salvo que estés desarrollando un proyecto personal que haces avanzar a ratos y por puro placer, lo habitual es que formes parte de un equipo de desarrollo; para trabajar en él, hay que dedicar tiempo y esfuerzo a coordinar con los compañeros, revisar el trabajo de otros y tareas similares.

Por otra parte, cuando programamos no podemos obviar que si nos consideramos desarrolladores profesionales, tenemos que respaldar gran parte de lo que hacemos con sus correspondientes pruebas; en ocasiones crear las pruebas adecuadas y ejecutarlas correctamente nos puede llevar más de la mitad del nuestro tiempo de trabajo. De acuerdo, están quienes tienen este rol, los testers, aunque lo cierto es que pocos equipos de trabajo tienen roles tan diferenciados, me temo.

¿Nos sentamos y nos ponemos a teclear código como locos? En absoluto, cualquier ejercicio de escribir algo nuevo lleva detrás un trabajo y esfuerzo de «diseño», en mayor o menor medida. Es verdad que tenemos el impulso de ponernos ante nuestro IDE cuanto antes, pero esto sólo lo podemos hacer para realizar prototipos y para «jugar en casa».

Cuanto mejor hagamos y desarrollemos todas esas actividades alrededor del acto de escribir código de producción mejor y de mayor calidad será este, sin ninguna duda.

Ahora bien, existe un elemento más y nada desdeñable: en nuestra actividad nos pasamos gran parte del tiempo depurando y corrigiendo errores. Esto es evidente, pero no lo suele ser tanto las implicaciones que conlleva.

Cuando uno lleva algunos años dedicados profesionalmente al desarrollo de software, se tiene la oportunidad de ver un poco de todo: obras geniales y también soluciones que terminan en la basura antes de ser completadas. Es realmente difícil determinar cómo valorar la calidad de un buen software: métricas de cobertura, diseños elegantes, etc. Aunque existen algunos

estándares, es algo todavía demasiado subjetivo. En mi opinión hay un elemento que no falla nunca.

La calidad de una pieza de código es directamente proporcional a la capacidad que presente para su depuración. Así de sencillo y, en mi modesta experiencia, esto no suele fallar.

Gran parte del tiempo lo pasamos corrigiendo errores, detectando fallos (nada peor que esos que surgen aleatoriamente), buscando defectos de diseño, etc. Por tanto, si esto es así, ¿puede existir una forma de escribir código que permita esa actividad de detección y depuración de errores mejor que otra? Por supuesto que sí, existe y es obligación de un desarrollador profesional escribir software «depurable», que sea fácil corregir.

Como en muchas ocasiones, lo que es de sentido común y evidente, algo para lo que todo el mundo parece estar de acuerdo, no lo es tanto cuando uno tiene que aplicarlo en su día a día. Escribir código ahora que sea fácil de depurar después, requiere de un esfuerzo continuo por seguir las buenas prácticas de desarrollo de software. El no hacerlo nos pasará una enorme factura en tiempo y esfuerzo cuando tengamos que ponernos a corregir errores (que sin duda habrá) o evolucionar el sistema. Digamos que aumentamos enormemente la deuda técnica al no escribir código fácilmente corregible.

En el equipo de desarrollo que ahora mismo lidero, nos ha pasado que a pesar de escribir muchísimas pruebas unitarias y de integración al mismo ritmo que escribimos código de producción, cuando hemos terminado un *sprint* de trabajo y a punto de cerrar una versión, las pruebas de validación nos han revelado muchos errores para los que hemos tenido que emplear bastante tiempo en resolver.

Si el software que tenemos que corregir es difícilmente depurable, el tiempo para enmendar errores será evidentemente muchísimo mayor. Nada más frustrante que pasarte una mañana entera buscando cómo resolver cierto bug que finalmente resulta que era una auténtica tontería…

Es inevitable en cierta medida introducir bugs en el código, aunque sí podemos minimizar el tiempo dedicado a depuración.

Una pregunta que me hago cada vez que doy por terminado la implementación de una tarea, requisito o historia de usuario es «todo esto que

he hecho, ¿lo podría depurar yo mismo o algún otro fácilmente dentro de unos meses?».

Existe una relación clara entre una solución que puede evolucionar y una solución para la que es fácil corregir o detectar errores. Su calidad intrínseca viene determinada por haber seguido o no ciertos principios de diseño y buenas prácticas de desarrollo de software.

En la literatura del desarrollo de software, hay en mi opinión tres libros que claramente sirven para aprender a escribir código más limpio y depurable, son «*Clean Code*» (de Robert C. Martin), «*Code Complete*» (Steven McConnell) y «*Refactoring: improving the Design of Existing Code*» (de Martin Fowler). Sobre ellos ya hemos hablado en algún momento en *El Libro Negro del Programador*; todo desarrollador profesional debe tener un ejemplar de cada uno de estos tres libros en la cabecera de su cama e, igualmente, debería haber una asignatura sobre estos temas en la titulación de ingeniería informática o cualquier otra en la que se enseñe a «programar».

Principios de diseño S.O.L.I.D., buenos mensajes y trazas de log, suficiente desacoplamiento entre las partes funcionales del sistema y los módulos del mismo, ausencia de código duplicado o con similitudes, funciones o métodos con relativamente pocos parámetros, elección correcta en los nombres de variables, clases, etc., hasta una buena estructura y organización del código en la solución. Todo, absolutamente todo, facilita la capacidad de resolver y depurar una aplicación. Defendemos la práctica y todo lo relacionado con el *clean code* precisamente porque nos da las habilidades necesarias, entre otras cosas, para generar código fácilmente depurable.

En cierta medida podemos decir que aprendemos a programar depurando errores, pero, ¿tenemos presente la capacidad de depuración de nuestro software mientras lo desarrollamos? ¿Tomamos alguna medida al respecto? ¿Lo tenemos realmente en cuenta en el día a día?

Las consecuencias de que nuestra aplicación no sea buena para poder depurarla son muchas (y todas ellas catastróficas); clientes insatisfechos cuando la corrección de algo se alarga demasiado en el tiempo, proyectos más costosos porque los bugs son difíciles y exigen mucho tiempo para su corrección, desarrolladores de software desmotivados por tener que dedicar

gran parte de su tiempo a solucionar errores (lo cual tiene también un precio en la productividad de los mismos) y un larguísimo etcétera.

A lo largo de los capítulos de este libro insisto mucho en que la clave del éxito para cualquier software es su mantenibilidad: el que sea fácilmente depurable (que sea fácil solucionar posibles bugs) es una parte más de este concepto tan poliédrico.

De nuevo vemos cómo la productividad de un desarrollador de software no tiene nada que ver con la cantidad de horas que pueda trabajar al día, sino con la capacidad de escribir software mantenible y depurable.

No podemos ser complacientes y dar por cerrado un módulo o tarea sin preguntarnos si hemos hecho todo lo posible para que el código que hemos generado sea lo suficientemente depurable o no. Si lo es, dejamos de acumular deuda técnica y podremos dedicar más tiempo a la incorporación de nuevas características y código útil al proyecto. Seremos, en definitiva, mucho más productivos y estaremos más motivados.

Puntos clave

- Resolver un problema desarrollando software no es suficiente; el código generado tiene que ser depurable.
- El esfuerzo de escribir código limpio y depurable es continuo y se debe hacer al mismo tiempo que se escribe código de producción.
- Pasamos gran parte del tiempo como desarrolladores corrigiendo errores, es algo intrínseco al software, razón de más para facilitar este trabajo, lo que hará que dediquemos más tiempo a escribir código de producción.
- Somos productivos en la medida en que somos capaces de escribir código que se puede seguir y corregir.

Esclavo de tu propia solución o cómo querer ser imprescindible

«A veces buscamos seguridad laboral dominando en exclusiva una solución de la que la compañía depende en cierta medida; esto nos hace cautivo de la misma y nos impide progresar profesionalmente. Si buscamos una zona de confort como desarrolladores de software, hemos errado en nuestra profesión.»

A lo largo de los años de vida laboral uno se va encontrando con multitud de personas, algunos se convierten en amigos, conocidos que van y vienen, con otros trabajas de cerca; en definitiva, terminas viendo un poco de todo (o eso creo aunque estoy abierto a descubrir nuevas sorpresas). Gente de lo más normal y gente totalmente friki.

Como desarrolladores de software, te encuentras también ciertos tipos de compañeros entre los que destacan algunos de especial pelaje. No sólo construimos software sino que además si tenemos cierta idiosincrasia psicológica, nos gusta apoderarnos de lo que hacemos, hacerlo nuestro y crear así nuestra especial «parcela de poder». Este comportamiento, que en principio puede parecer que no es más que una postura entendible para defenderse laboralmente, es más perjudicial que positivo para los intereses de un buen desarrollador.

Hasta tal punto es así, que en ocasiones te encuentras con quien dentro de un ámbito laboral, se niega a «darte información» para hacer funcionar algo o mantiene en una caja negra algo que sólo esa persona entiende o puede mantener. Esto debería ocurrir de manera puntual y, sin embargo, esta postura es relativamente común. En otras ocasiones y por diversas

circunstancias, eres tan bueno y eficiente en algo concreto que se te encasilla en un rol que te convierte en prisionero del mismo y que impide cualquier progreso laboral y profesional.

Este es uno de los mayores errores que he cometido como desarrollador de software. Hace algunos años comencé a conocer un producto muy específico de una compañía norteamericana; con el tiempo, terminé convirtiéndome en experto de este producto en el que además estaba basada la solución final en la que trabajábamos entonces. Era algo extremadamente específico dentro del sector de la telegestión de contadores electrónicos domésticos (o *smart meters*).

Al cabo de uno o dos años, lo conocía tan bien que incluso ayudaba al equipo de desarrollo que lo evolucionaba y mantenía a solucionar y detectar errores. Hasta aquí todo perfecto, aunque no me estaba dando cuenta de algo fundamental que iba surgiendo a mi alrededor al gozar de todo ese conocimiento sobre ese producto y en cierta medida que dependieran de mí en la solución final. Esa «dependencia» no era en absoluto sostenible.

De manera natural, en mi entorno laboral se me asoció rápidamente como el máximo experto del producto que, además, lo había sabido usar con solvencia y suficiente éxito en el proyecto para el que trabajábamos. Para cualquier asunto relacionado siempre acudían a mí para resolverlo. Hablar de ese producto era hablar de mí dentro del departamento: se me había encasillado completamente como experto de esa tecnología. Lo que a primera vista parece ser bueno, a medio plazo no lo es tanto.

Pasaron más de dos años y comencé a sentir cierta presión a nivel personal ya que veía cómo se me pasaban de largo ciertas tendencias tecnológicas para las que apenas tenía tiempo de conocer; durante ese tiempo, mi dedicación en torno a ese producto y la solución final basada en él que desarrollamos era total y no tenía margen para hacer «otro tipo de cosas»; lógicamente tampoco me dejaban ya que mi dedicación a ese proyecto era demasiado crítica. Mi especialización en él me estaba haciendo estar completamente cautivo.

Comencé a preguntarme qué significaba esto a nivel profesional: ¿era bueno que se confiara en mí tanto para esa área? ¿Me hacía tener más seguridad laboral dentro de la empresa? ¿Estaban pasando de largo algunas

oportunidades? ¿Prefería seguridad o estaba dispuesto a pagar algún precio para probar otras cosas? ¿Era esa «seguridad» real?

Al mismo tiempo que me hacía estas preguntas, intuía que estaba perdiendo el tren en otro tipo de proyectos y responsabilidades: mis jefes me tenían tan asociado a esa área tan específica que era demasiado arriesgado dársela a otros, al menos esa era la impresión que me daba.

Por un lado mi éxito en el uso de esa tecnología me había hecho cautivo de la misma, perdiéndome otras oportunidades laborales y profesionales, así de sencillo; así es como lo veo cuando ha pasado mucho tiempo desde entonces aunque en ese momento todo eso se traducía en insatisfacción laboral y un estrés añadido. Por otro lado, en momentos en que lo veía todo negro tenía una falsa sensación de seguridad pensando que la compañía no podría prescindir de mí en ningún momento. Si algo he aprendido hasta ahora, es que no hay nadie absolutamente imprescindible en ninguna compañía.

No obstante, esa situación que yo vivía como negativa para mi desarrollo profesional, suele ser entendida como deseable y hasta perseguida por algunos desarrolladores de software. Craso error.

El razonamiento, un poco retorcido, suele ser el siguiente: puesto que soy el único que controlo «esto», porque lo conozco mejor que nadie o porque lo he hecho yo mismo y sólo yo soy capaz de ponerlo en marcha, la empresa nunca va a prescindir de mí y, además, mi ego se encumbra entre mis compañeros porque soy el único que «controla y conoce» esto y aquello, sientes que «dependen de ti». Ahora veo claramente que este es un patrón de conducta que han repetido algunos de mis compañeros a lo largo ciertos proyectos en los que he trabajado.

En mi opinión esto no es bueno y es un completo error: un desarrollador de software no puede buscar en ningún momento la «zona de confort» que uno siente cuando conoce perfectamente una tecnología o producto y de la cual vive y que, tarde o temprano, terminará en una completa obsolescencia. Un desarrollador de software está obligado a moverse al mismo ritmo que las tendencias tecnológicas y, en la medida de lo posible, debe participar en distintos proyectos a lo largo del tiempo, nada peor que trabajar en el mismo proyecto demasiado tiempo. Estamos hablando de que en tecnologías

software hace cinco años era la prehistoria.

Dejando a un lado el diletantismo tecnológico que ya hemos tratado en un capítulo anterior, ¿es razonable que un informático desarrollador de software ignore hoy día tendencias como la virtualización, *cloud*, *big data*, bases de datos no sql, etc.? Puede que sí, pero entonces nos estamos dando unas alas muy cortas y nuestra progresión profesional se reducirá drásticamente.

Cuando queremos vivir de un nicho muy pero que muy concreto, como fue mi caso en su día y que he puesto de ejemplo con total honestidad, en realidad lo que estamos haciendo es buscar una parcela de seguridad, por un lado laboral y por el otro reducir la ansiedad que puede provocar el enfrentarte a cosas nuevas. Este comportamiento no se lleva nada bien en un sector en el que todo, absolutamente todo, viene definido con una obsolescencia programada de antemano, con muy reducidas excepciones.

Un buen desarrollador de software lleva escrito en su ADN la capacidad de trabajar en muchos proyectos distintos, en pivotar de tecnologías cuando haga falta y la capacidad de reinventarse para adaptarse a las nuevas circunstancias profesionales. Si bien se dice que el mundo laboral que nos espera es el de cambiar en varias ocasiones la compañía para la que trabajamos a lo largo de nuestra vida profesional, ¿por qué no entendemos igualmente que cambiaremos durante todo ese tiempo de tecnologías, paradigmas, metodologías y formas de hacer e implantar los proyectos?

Se me podrá objetar que todavía hay quienes se dedican a mantener sistemas en Cobol, y así es, pero yo personalmente me deprimiría si me dijeran que tendría que pasar los próximos diez años de mi vida laboral haciendo eso exclusivamente, por poner un ejemplo, aunque esto es una valoración muy personal.

La tecnología (y por qué no, también la sociedad) avanza con pioneros que arriesgan a proponer cambios disruptivos en una u otra área, en mayor o menor medida; ahí es donde están las oportunidades, no persiguiendo un sueldo a final de mes manteniendo algo que conocemos hasta la saciedad pero que nos hace sentir seguros. La seguridad no es un hecho, no existe por sí misma, es una «sensación» personal.

Las oportunidades se nos presentan con el movimiento: en nuestra actividad esto se traduce en trabajar en distintos proyectos y con diversas

tecnologías e igualmente en distintas compañías con diferentes enfoques laborales. No digo que haya que cambiar continuamente porque sí, sino que debemos identificar cuándo una dedicación exclusiva a algo termina lastrando nuestras oportunidades laborales y de progreso profesional, sobre todo cuando queremos cambiar a otro tipo de cosas.

Los mejores profesionales con los que me he encontrado han cambiado de compañías en distintas ocasiones, lo cual no implica que quienes llevan mucho tiempo en la misma tengan más probabilidad de ser malos profesionales, pero sí revela un síntoma: un buen profesional se trata a sí mismo como una empresa, buscando el mejor proyecto para su desarrollo, en la compañía A, en la B o como *freelance*.

El tratar y gestionar la propia carrera profesional de uno, como si fuese una compañía, es un concepto que me chocó bastante cuando leí sobre él por primera vez en el libro «*MBA personal*» (de Josh Kaufman), libro que recomiendo encarecidamente.

También se puede objetar que en tiempos de crisis uno se agarra a un clavo ardiendo, y así es, como es lógico, aunque destaca la falta de movilidad ya no entre empresas, sino entre departamentos de la misma compañía, al menos en los entornos en los que yo me he movido hasta ahora.

Los paradigmas laborales están cambiando, «por si no os habíais dado cuenta»; las compañías ya saben que es más eficiente tener contratados al personal estructural justo y mínimo imprescindible y subcontratar puntualmente a autónomos o *freelancers*, lo que quiere decir que trabajaremos más por proyectos y por resultados. Así son las cosas y en este esquema no pinta absolutamente nada querer aferrarnos a algo concreto para no dejar de ser imprescindibles. Hoy día, y esto es más cierto en software, no hay nadie imprescindible: sólo es cuestión de que alguien al menos igual de avispado que tú y con algo de tiempo por delante alcance el mismo nivel de conocimiento y *expertise* que tú en un producto concreto o tecnología, cuando no la compañía lo tira literalmente a la basura y lo sustituye por algo parecido, lo cual seguramente sería más barato a medio plazo que mantenerte a largo plazo.

Si esto es así, al menos lo es para mí, se presentan implicaciones interesantes: comenzamos un proyecto sabiendo a priori que en algún

momento, tras su finalización y pasado un tiempo de su puesta en marcha en producción, lo abandonaremos para dirigirnos a otro proyecto dentro de la misma compañía o fuera. No es que el proyecto se cierre o deje de continuar, el proyecto no es el que se abandona, sino que somos nosotros los que le abandonamos: continuará pero con otra gente que llegará para asumir la misma o similar responsabilidad que tú tenías.

Por tanto, el código que escribimos será asumido por un compañero o compañera que seguramente ni conozcamos, para lo cual será necesario cumplir con parámetros de mantenibilidad; ésta no sólo consiste, como insisto hasta la saciedad en *El Libro Negro del Programador*, en que un software tenga la capacidad de ser evolucionado de manera sencilla, también es que cualquier persona que no sepa nada del mismo sea capaz sin demasiado esfuerzo de evolucionarlo y mantenerlo; para ello, debe entender fácilmente cómo están hechas y estructuradas las cosas en el proyecto: buena documentación, coherencia de diseño, código limpio, ver claramente el *design sense* (como diría Robert C. Martin) y un larguísimo etcétera.

Puntos clave

- Es un error hacernos imprescindibles de una aplicación crítica para la empresa: nos convierte en cautivos de la misma e impide nuestro progreso.
- Sólo cambiando periódicamente de proyectos encontraremos mejores oportunidades.
- Para un desarrollador de software con inquietudes de progreso, realizar las mismas tareas durante mucho tiempo es antinatural.
- El sentirnos «seguros» se debe basar en nuestra solvencia tecnológica y en una alta autoestima para saber adaptarnos a nuevos proyectos y retos, no en dominar en exclusiva cierta parcela importante para una empresa.
- Comenzamos un proyecto sabiendo a priori que en algún momento saldremos de él; por tanto, nos esforzaremos en que otros lo puedan asumir fácilmente.

Aprendemos de otros (o cómo no ser un desarrollador perezoso)

«Aprendemos desarrollando proyectos, lo que es obvio, lo que no lo es tanto es que aprendemos y afinamos nuestras habilidades para resolver proyectos leyendo y analizando los realizados por otros desarrolladores.»

Es frecuente que en nuestro día a día estemos tan absorbidos en el desarrollo del software que estamos generando que poco tiempo nos queda para aprender otras tecnologías o «echar un vistazo» a otro tipo de proyectos. Así las cosas, es relativamente común que pasemos demasiado tiempo avanzando en el mismo proyecto.

Podemos leer un buen manual sobre cierta tecnología, de arriba abajo y muy concienzudamente y, no obstante, no tener ni idea de cómo aplicarla en un proyecto real; este conocimiento sólo lo da la experiencia y a diferencia de otras profesiones aprendemos fundamentalmente «viendo y estudiando» el código generado por otros pero, sobre todo, trabajando directamente en proyectos. Podemos afirmar que somos expertos en C# pero si no tenemos el bagaje de varios proyectos con ese lenguaje, nuestra credibilidad se pierde en un agujero de desagüe…

Unos de los mayores círculos viciosos en los que solemos incurrir los desarrolladores de software es dedicarnos casi en exclusiva y durante mucho tiempo al trabajo en un mismo proyecto, donde la arquitectura y la forma de hacer las cosas, bien o no tan bien, están ya más que establecidas. Vamos evolucionando el proyecto con nuevas características pero siempre «dentro del mismo».

He vivido esta situación en varias ocasiones y una de las consecuencias negativas es que terminas adoptando una forma extremadamente rígida de hacer las cosas: cuando comienzas un nuevo proyecto, tiendes a aplicar aquello que mejor conoces, que no es más que el modo en el que has pasado más tiempo trabajando en el proyecto anterior. De ese modo repites sin darte cuenta ciertos patrones de arquitectura, las mismas formas de organizar la solución, etc. No es que esté mal, si eres un desarrollador profesional lo eres porque haces todo eso relativamente bien, pero, ¿estamos seguros de que no podríamos hacer las cosas aún mejor?

¿Podría un buen escritor de novelas escribir sus obras sin leer absolutamente nada de otros autores? ¿Podría un pintor crear su propio estilo sin haber bebido de las fuentes de distintas escuelas artísticas? ¿Podría un arquitecto mejorar un diseño sin basarse en los que otros han creado? Por tanto, ¿podemos los desarrolladores de software mejorar nuestro trabajo sin estudiar y ver cómo otros desarrolladores (seguramente más expertos y listos) han resuelto cierto tipo de problemas?

Aunque en nuestra profesión se insiste mucho en el cuidado del diseño, lo cual es muy importante, se obvia a menudo la realidad de que aprendemos fundamentalmente leyendo ejemplos y código hecho por otros. Así de sencillo y, sin embargo, cuántas personas me he encontrado que sólo y exclusivamente conocen bien los proyectos en los que han trabajado y que, fuera de ellos, no sabrían ni por donde comenzar uno nuevo; o bien, resuelven todos los problemas que se encuentran buscando en foros o abusando de Stack Overflow. Este es un perfil de desarrollador bastante común, llamémosle el «desarrollador perezoso», aquel que fuera de un proyecto particular o tipo de proyectos no sabría ni por donde comenzar.

El primer problema de este «acortamiento de miras» es que sueles arrastrar los vicios y las formas no tan buenas de proceder para resolver ciertas situaciones; por poner un ejemplo, puede que estés acostumbrado a usar una única librería de creación de mensajes de log e ignoras que existen varias mucho más consolidadas y mejores. Puede que tareas tan sencillas como leer un simple archivo lo suelas hacer de la forma en la que estás acostumbrado, hasta que un buen día y casi por casualidad, encuentras que hay una forma, sencilla, elegante y eficaz de hacer lo mismo con la mitad de

líneas de código. O bien estás acostumbrado, porque en cierta época se hacía así, a desarrollar capas de integración basadas en servicios web y ahora descubres que para ciertos escenarios es mucho más conveniente (y fácil) implementar servicios REST. ¿Cómo ibas a descubrirlo sin ver el código hecho por otros en foros, proyectos o artículos?

Tengo que reconocer que en ocasiones he notado la misma sensación que los escritores describen como el «miedo a la página en blanco» cuando he tenido que comenzar un proyecto desde cero. ¿Cómo organizamos la solución? ¿Por qué áreas comenzamos en primer lugar? ¿Qué arquitectura planteamos en un principio? Es un momento crítico porque de la correcta respuesta a estas preguntas dependerá en gran medida que el desarrollo del proyecto vaya como la seda o sea difícil avanzar en él.

Pero, ¿cómo aprendemos este tipo de cosas? Podemos leer un buen manual, uno de los más reputados, sobre cierto lenguaje y tras terminarlo apenas podremos hacer poco más que los ejercicios y ejemplos que contiene. Aprendemos desarrollando proyectos, lo que es obvio, lo que no lo es tanto es que aprendemos y afinamos nuestras habilidades para «resolver proyectos» leyendo y analizando los realizados por otros desarrolladores; para ello hay que tener cierto grado de humildad.

Del mismo modo que un escritor descubre recursos estilísticos leyendo libros de otros autores, un desarrollador de software encuentra ciertas formas prácticas de hacer inimaginables, muchas veces con grata sorpresa.

Por poner un ejemplo, nunca se me habría ocurrido utilizar el estilo de llamadas encadenadas que se realizan en jQuery; cuando descubrí esta librería, además de quedarme maravillado por su sencillez y elegancia, descubrí también que para ciertos escenarios era una buena técnica el estilo de llamadas encadenadas, lo que me ha permitido aplicarlo en más de una ocasión.

Cada uno tiene como su propia impronta o forma de escribir código; es curioso pero en la plataforma en la que trabajo ahora mismo y en la que participan cinco personas, casi puedo adivinar exactamente quién ha hecho qué sin mirar el historial de control de cambios… Cinco personas participando en el proyecto, cinco estilos distintos de resolver los problemas aunque se hayan definido algunas guías de estilo.

Un buen desarrollador de software no vive centrado exclusivamente en los proyectos en los que trabaja, sino que como parte de su proceso continuo de formación, debe leer y analizar proyectos realizados por otros. De esa manera se aprenden estilos, trucos y detalles que difícilmente uno podría llegar a aprender.

Para ello uno tiene que enfocar su trabajo con cierta humildad y estar abierto a que haya siempre alguien que te pueda señalar cómo mejorar esto o aquello. Los mejores desarrolladores que he conocido han sido los más humildes con su propio trabajo, los más abiertos a reconocer que hay soluciones mejores que las que uno ha propuesto. Yo siempre digo que ojalá me señalaran mis errores a diario: así aprendería mucho más rápido y mejor. Agradezco a todos los que me enseñan a hacer las cosas mejor.

Vivimos en un entorno abierto altamente cooperativo, con enormes recursos compartidos y, sin embargo, muchas veces nos lanzamos a resolver proyectos «con la primera idea que se nos ocurre». Ese es el primer error del desarrollador perezoso.

No hay nada mejor en tu propio currículum que incluir una lista pormenorizada de los proyectos y trabajos en los que has participado: esa heterogeneidad dará una idea clara de tu grado de experiencia y versatilidad.

Puntos clave

- Es importante (yo diría que hasta imprescindible) leer de vez en cuando proyectos realizados por otros. Nos puede ayudar a encontrar formas mejores de resolver problemas.
- Deberíamos participar en un número amplio de proyectos a lo largo de nuestra experiencia laboral; aprendemos y mejoramos como profesionales participando en proyectos heterogéneos.
- En ocasiones aprender sobre otras tecnologías distintas de aquellas que usamos habitualmente nos permite conocer otros modos y técnicas de hacer las cosas que podremos incorporar al lenguaje o framework que utilizamos en mayor medida.

Potenciando la productividad con la gestión de la configuración e integración continua

«La gestión de la configuración y la integración continua nos sirven para trabajar productivamente: como profesionales, debemos siempre intentar trabajar de manera eficiente. El mejor profesional no es el que trabaja más, sino mejor.»

Cuando hablamos de desarrollar software, no nos referimos exclusivamente a sentarnos delante de un ordenador y escribir líneas de código ni mucho menos. A lo largo de *El Libro Negro del Programador* hemos visto infinidad de elementos que interfieren, afectan y permiten que el acto de programar sea un auténtico desastre o se generen las condiciones para producir software de calidad al tiempo que el desarrollador utiliza su tiempo de manera altamente productiva. Si esto aún no ha quedado claro, sugiero volver a comenzar por el capítulo uno. La cuestión es que normalmente esta idiosincrasia del software no la suelen entender bien las capas de decisión; lo peor es que en ocasiones ¡ni los propios desarrolladores!

En un tiempo en el que parece que se encomia el hecho de trabajar más y más, yo siempre digo que no se trata de dedicar muchísimas horas al trabajo, sino de ser productivos durante el tiempo que estamos trabajando, que no tiene nada que ver. También hemos visto claramente que precisamente el hecho de dedicar muchas más horas de manera crónica, día tras días, en nuestro contexto laboral, cualquiera que sea este, implica seguramente fallos de organización y que el trabajo se hace seguramente de manera improductiva: la productividad no depende de trabajar más horas sino mejor. En palabras del genial Raimon Samsó (en su libro «El código del Dinero»),

nos movemos hacia un nuevo paradigma laboral en el que el éxito vendrá de la mano del talento, la tecnología, la innovación y, sobre todo, el reinventarnos a nosotros mismos continuamente. En nuestra profesión es más fácil que en otras poner estas habilidades en marcha. Lo fundamental de todo es que hay que cambiar el chip y no pensar en «llenar de trabajo» las ocho horas de la jornada laboral o las que sean, sino que tenemos que pensar en cumplir un objetivo y alcanzar unos resultados: no nos van a pagar por horas, sino por resultados; para ello tenemos que trabajar productivamente.

Las herramientas imprescindibles de un buen equipo de desarrollo que se considere altamente productivo son una buena gestión de la configuración y la integración continua, por tanto, de la buena implantación que hagamos de ambos conceptos dependerá en cierta medida el conseguir más o menos éxito en el proyecto.

La gestión de la configuración en software (*software configuration management*) consiste en la definición de un conjunto de procesos que apoyen la producción de calidad en cada una de las etapas de un proyecto software, lo cual implica un uso correcto de la herramienta de control de versiones, la definición de la forma en que se van a generar y congelar versiones, cómo se van a gestionar los bugs, etc. De acuerdo, esto suena demasiado académico: dicho muy básicamente, la gestión de la configuración no es más que establecer cómo vamos a generar y controlar las versiones que se generen del producto software.

¿Se piensa en la definición de la gestión de la configuración al inicio del proyecto? Casi nunca, me temo y, sin embargo llegará el momento durante el mismo en que se tendrán que plantear preguntas que debieron quedar resueltas al comienzo, como por ejemplo, ¿cómo congelamos esta versión antes de ponerlo en producción en el cliente para seguir avanzando en una revisión menor? Este tipo de cosas se suelen resolver «sobre la marcha», me temo, lastrando la calidad del proyecto y la productividad del trabajo en el mismo.

Cuando sí existe una gestión de la configuración definida, se suele arrastrar aquella «forma de funciona» que teníamos para otros proyectos anteriores, sin darnos cuenta de que cada proyecto, según su naturaleza, número de miembros en el equipo, número de despliegues en producción

estimados, etc. requiere de su propia y específica gestión de la configuración.

Si no se definen este tipo de cosas, perderemos muchísimo tiempo a lo largo del proyecto y la calidad de lo generado se resentirá, así de sencillo y contundente. Somos más productivos si al inicio de un nuevo trabajo nos paramos un momento a definir, precisamente, cómo va a ser esta gestión de la configuración.

En el equipo en el que trabajo actualmente, tenemos establecido una simple norma que indica cómo «subir» (*check-in*) cualquier avance en nuestras tareas de desarrollo: antes, actualizamos el proyecto desde la versión de código fuente, compilamos, si no hay problemas ni conflictos, entonces «subimos». Estos sencillos pasos reducen enormemente la cantidad de conflictos en el código cuando trabajan varias personas en el mismo proyecto. Esto es gestión de la configuración, las normas básicas para trabajar, pero pensando siempre en reducir los problemas potenciales para centrarnos en lo que aporta valor al proyecto.

En el mejor de los casos, uno de los miembros del equipo es el que se dedica a hacer el seguimiento oportuno para que se trabaje correctamente según la gestión de la configuración definida.

Suele suceder que cuando se inicia un nuevo proyecto, cuando estamos todos ansiosos por comenzar a ver algo funcionando, veamos como un gasto inútil sentarnos y dedicar un tiempo a planificar una correcta y específica gestión de la configuración. Peor aun cuando la mayoría de los proyectos comienzan ya retrasados y con prisas…, más difícil será entonces el sentarse un tiempo a establecer las normas de funcionamiento. Las prisas, cómo no, siempre provocan que al final las cosas, paradójicamente, tarden más en realizarse y con peor calidad.

No deberíamos comenzar un nuevo proyecto sin establecer claramente cuál va a ser la gestión de la configuración para el mismo. Esta es una regla fundamental que hay que seguir y es sorprendente el poco tiempo y esfuerzo que lleva sentarse unas horas y poner por escrito la estrategia a seguir.

El mayor desastre en este aspecto que he conocido fue un departamento que desarrollaba software para dispositivos embebidos, con cientos de clientes repartidos por medio mundo para los que averiguar qué versión de qué producto, con qué cambios, etc. era una tarea infernal; en ocasiones no se

podía averiguar qué fuentes correspondía a qué binarios para un cliente en particular; ni que decir tiene que los que trabajaban allí vivían permanentemente estresados. Todo eso se habría solucionado en el principio de los tiempos si se hubiera establecido una gestión de la configuración correcta. ¿Cuántas horas improductivas se dedicaban a resolver ese tipo de problemas? ¿Cuántos clientes insatisfechos? ¿Cuánto esfuerzo (dinero) malgastado? Detrás de la productividad siempre está el trabajo bien organizado y el saber hacer; al final de la cadena, gracias a la productividad generamos más ingresos con menos esfuerzo.

Por su parte, ¿cuántas horas perdemos en resolver conflictos de código porque algún miembro del equipo ha subido algunos cambios hechos hace una semana?, por poner un sencillo ejemplo.

¿Cuánto tiempo ha permanecido «roto» el proyecto a nivel de compilación impidiendo el buen avance del mismo?

¿Cómo podríamos haber detectado cuanto antes que ciertas partes del proyecto fallan cuando funcionan juntas, cuando se integran unas con otras?

¿Cómo podemos saber lo antes posible que el proyecto ha dejado de funcionar o que alguna prueba ha dejado de ejecutarse con éxito?

Todo esto, que antiguamente se hacía de manera más o menos manual (si es que se hacía) lo cubre el concepto de integración continua. En la esencia de la integración continua está el reducir al mínimo este tipo de problemas, no perder tiempo en conflictos que se pueden resolver por sí solos con un buen procedimiento de trabajo, asegurar que la compilación siempre es correcta, lo que es fundamental para el equipo pueda avanzar en paralelo, y, sobre todo, detectar lo antes posible cuándo algunas pruebas han dejado de funcionar, ya que cuanto más se retrase su resolución, más tiempo llevará. Productividad, productividad y productividad, esto es integración continua.

Al igual que la gestión de la configuración, se ponen las bases de la integración continua al inicio del proyecto; de su buena aplicación y puesta en marcha dependerá que perdamos decenas o cientos de horas en resolver problemas que de otro modo no hubieran llegado a producirse. No en vano se suele decir que el mejor problema es aquel que no se produce nunca o aquel que se resuelve solo.

La integración continua también asegura una mejor calidad del código, ya

que al detectar pruebas fallidas, nos permite resolverlas cuanto antes y nos obliga a escribir desde el principio código correcto.

Si trabajamos en un proyecto personal en el que nosotros somos los únicos que lo desarrollamos, lógicamente puede que no tenga sentido poner en marcha una gestión de la configuración con mucho detalle y tampoco un mecanismo de integración continua. Para un equipo de dos o más personas es imprescindible poner en práctica estos mecanismos y, sinceramente, para equipos grandes no veo otra manera de trabajar coordinadamente sin estas herramientas fundamentales.

Me temo que en ocasiones el gestor del proyecto sólo percibe el avance del mismo según el número de líneas de código escritas o tareas de desarrollo completadas, sin ver que la definición y correcta implantación de la gestión de la configuración e integración continua acelerará enormemente el trabajo y permitirá que los desarrolladores estén la mayor parte del tiempo, ¡programando!

Comenzar a trabajar según los procedimientos de la gestión de la configuración definidos y desarrollar nuestras tareas de código en consonancia con la integración continua puede ser algo chocante para quien ha trabajado siempre «a su manera», con los procedimientos «en la cabeza» y sin una cultura clara de creación de pruebas que demuestren que todo lo que hacemos funciona en todo momento. Una vez que damos el paso e incorporamos a nuestra cultura de desarrollo de software estos conceptos, llegamos a un punto de no retorno en el que no se nos ocurre volver atrás en ningún momento, mejorando así como profesionales. Como se suele decir, una vez que lo pruebas, imposible volver atrás.

Puntos clave

- No somos productivos por trabajar muchas horas, si no por hacer el mismo trabajo y con la misma o mayor calidad en menos tiempo. El presentismo en el trabajo es un problema, pero como desarrolladores profesionales sólo debemos buscar trabajar eficientemente y buscar los mejores resultados.
- Un desarrollador de software es aún más profesional si se siente cómodo

y sabe trabajar bajo el paraguas de una correcta gestión de la configuración e integración continua.

- Si aterrizamos en un nuevo equipo de trabajo en el que no está asentada esta cultura aún, debemos indicarlo, por el bien de todos…
- La integración continua nos aleja de la nefasta proporción 20/80 (80% del tiempo resolviendo errores y 20% escribiendo código nuevo).

La era del emprendimiento |

«Como desarrolladores de software tenemos más a mano que nadie las herramientas y el conocimiento para poner en pie nuevos proyectos e ideas porque dominamos los átomos de la nueva economía que lo está cambiando todo.»

Desde hace muchos años intento combinar mis responsabilidades laborales con la puesta en marcha de proyectos de diversa índole y naturaleza. Algunos de corte tecnológico, otros, digamos, de un perfil más tradicional. Con el tiempo, algunos se quedan en simples intentos frustrados y otros tienen éxito, pero lo que sí es cierto de todos ellos es la extraordinaria riqueza personal y profesional que te aporta el participar activamente en la materialización de ideas y proyectos: al final, terminan enriqueciendo enormemente tu día a día en cualquier actividad.

El extraer posibilidades de aquí y de allá para llegar a algo útil se convierte en definitiva en un hábito, en una capacidad de «saber ver» lo que otros ignoran por falta de coraje, miedo al fracaso o, sencillamente, comodidad. No hace mucho leí a un reputado economista que decía que estamos pasando de la economía de la comodidad a la economía del emprendimiento: ahora tenemos que defender nuestra propia marca personal para conseguir un proyecto temporal y crearte una buena reputación profesional en la red.

Los paradigmas laborales están cambiando, lo hemos repetido a lo largo de *El Libro Negro del Programador*; llegamos a una nueva forma de trabajar, de aportar valor, de sinergias entre equipos, de nuevas profesiones que hace diez años no existían, de presupuestos económicos por resultados y no por

tiempo, viejos y tradicionales sectores desaparecen o se reinventan con una piel completamente diferente.

En un informe reciente de una gran compañía de servicios sobre perspectivas laborales para 2014, se identifican hasta treinta nuevos tipos de empleo que hace diez años no existían y que serán demandados de aquí al año 2030. Es decir, los tipos de trabajo evolucionan y mucho, si no evolucionas con esa demanda, estarás fuera del mercado.

Gran parte de ese cambio brutal se debe a las tecnologías de la información en donde los desarrolladores de software protagonizamos un papel fundamental.

Hay quienes perciben ese cambio, ese salir de nuestras zonas de confort, como una terrible amenaza y viven con inquietud y alarma constantes; pero también hay quienes lo ven como una maravillosa oportunidad para la que el talento, la creatividad y la profesionalidad son esenciales. Yo digo siempre que pasamos del presentismo al talentismo, de lo rígido y de que te den las cosas hechas a lo flexible y a tener que encararlas tú. Si esperas encontrar ese trabajo donde te digan exactamente qué hacer en todo momento, tienes que saber que escaseará cada vez más en el futuro.

No me considero autorizado para evaluar si todo esto es bueno o malo; imagino que dependerá según la especial idiosincrasia y contexto de cada uno, pero de lo que sí estoy seguro es que nunca ha habido más oportunidades como ahora para atrapar el éxito:

- Las barreras de entrada en muchos negocios se han reducido o han desaparecido notablemente.
- Muchas actividades tradicionales y permitidas sólo a ciertos círculos elitistas se han democratizado; por poner un ejemplo, ¿se necesita hoy una editorial para publicar un libro? De ninguna manera, existen opciones al alcance de cualquiera para dar a conocer tu trabajo e, igualmente, puedes contratar un profesional editor o un traductor directamente para que te ayuden con tu proyecto.
- Del mítico garaje polvoriento de los primeros geeks hemos pasado a poder publicar un blog de contenidos casi «a golpe de clic» o poner en marcha una idea totalmente innovadora a un coste irrisorio, o bien

encargamos a otro, en la otra punta del mundo, la creación de un prototipo.

- Igualmente evolucionan hasta las fuentes clásicas de financiación: de la obsoleta visita a un banco de toda la vida pasamos a tener la posibilidad de ponernos en contacto con *business angels*, participar en campañas de *crowdsourcing* o buscar emprendedores y socios con inquietudes similares, todo a través de la red.

Si la idea es buena, nunca ha habido más posibilidades de poner en marcha las herramientas para materializarla.

Los desarrolladores de software tenemos la suerte de ver este movimiento frenético «desde dentro» ya que muchos de los proyectos en los que se innova actualmente tienen un fuerte componente tecnológico, ya sea porque su ambiente natural es la red, porque como base necesita al menos de un portal que le dé cobertura o porque sea cual sea la naturaleza del proyecto es impensable que se haga sin el uso directo o indirecto de aplicaciones software más o menos especializadas.

No obstante, ¿aprovechamos igualmente esta nueva economía de la oportunidad basada en las nuevas tecnologías?

No se necesita tener la infraestructura de una gran compañía para poner en marcha proyectos que pueden funcionar y que incluso pueden cambiar las reglas del juego. Dentro de unos años casi nadie hablará, seguramente, de WhatsApp y, sin embargo, hoy lo usan 450 millones de personas. *Bloggers* se ganan un ingreso extra escribiendo y opinando sobre lo que más les gusta. Neorurales se van de una gran ciudad a un pequeño pueblo para vivir según sus inquietudes, viviendo de una web donde comparten y escriben sobre alimentación, vida sana y terapias alternativas (El Blog Alternativo). Un albergue rural tiene siempre puesto el cartel de completo en su puerta porque con poco esfuerzo, aseguran los clientes desde su propia web con una inversión mínima en marketing y SEO. Ordenamos transferencias desde nuestro propio teléfono móvil. Otros se centran en «conectar» personas con multitud de aplicaciones. El mundo *cloud* nos permite trabajar de otra manera en entornos más escalables, baratos y seguros.

El libro de Chris Guillebeau y de título «100€ startup» está lleno de

ejemplos de este tipo: yo siempre digo que no se trata de lanzar un cohete a la luna, sino de tener los pies en el suelo, pensar e intentar poner en marcha una idea con disciplina, esfuerzo y perseverancia.

Este libro es uno de esos proyectos personales que comencé hace año y medio. En esencia, el hilo conductor de todos los capítulos anteriores es el de aprender a trabajar mejor en nuestra profesión para poder centrarnos siempre en lo que realmente aporta un valor final, no malgastando nuestro tiempo y esfuerzo en tareas de relleno e improductivas. Nada peor que te paguen por un tiempo improductivo o por un trabajo que se tira a la basura. Es un ejemplo también de las posibilidades de la «sociedad del conocimiento» y las «nuevas tecnologías de la información»: su publicación directa, sin pasar por el filtro de una editorial tradicional que te seleccione de entre miles de otros libros es un ejemplo de cómo muchas actividades están mudando de piel.

¿Nos damos cuenta de que los desarrolladores de software no «sufrimos» esta nueva economía sino que nuestro trabajo es parte consustancial de la misma? Aprovechemos más que nadie esta circunstancia para aportar a la sociedad nuestro grano de arena en forma de ideas y proyectos para hacerla más humana y mejor; al final, eso es lo que siempre pretende nuestro trabajo: aporta valor útil a nuestros usuarios finales.

Conclusiones

Este libro nació con la idea de responder a la siguiente pregunta: ¿qué distingue a un profesional desarrollador de software de otro cuya carrera está llena de tropiezos, frustraciones y trabajos de mala calidad?

Responder a esa pregunta ha sido mi humilde pretensión a partir de la experiencia que he acumulado en los últimos años trabajando directamente para compañías y poniendo en marcha diversos proyectos emprendedores. Curiosamente, muchas de las claves son igualmente aplicables a otros campos que nada tienen que ver con el software: motivación, entrega, disciplina, buena organización, uso correcto de las herramientas que tenemos a nuestra disposición, colaboración y, sobre todo, mucha pasión y vocación por lo que más te gusta hacer.

Escribir un buen proyecto software, con calidad, que cumpla sus objetivos (lo que implica una revisión constante de requisitos, especificaciones o historias de usuario) se asemeja más a una maratón que a una carrera corta de velocidad: mientras que la primera es larga y llegamos a la meta con un ritmo y esfuerzo más o menos constantes, la segunda es más rápida, sí, pero improvisada y frenética para llegar exhaustos al final como sea.

Desde un punto de vista no técnico, pero sí muy humano, los mejores desarrolladores de software que he conocido (y aquellos que no conozco personalmente pero sí a través de sus libros y artículos) se distinguen claramente al mostrar una pasión extraordinaria por lo que hacen, por el «amor» que llegan a poner en una solución elegante, fina, funcional, de utilidad para los usuarios finales. Nos pueden pagar por hacer un trabajo cualquiera como desarrolladores; no obstante, es innegable que si no

sentimos pasión o no creemos en el proyecto en el que nos embarcamos difícilmente vamos a conseguir un resultado mejor: el proyecto se hará, pero «mercenariamente» y con muchas deudas técnicas (que al final se traducen siempre en costes extra, un equipo quemado y un producto o proyecto más caro).

Es incuestionable que un mal estilo de gestión influirá enormemente en el equipo de desarrollo que se encarga de ejecutar un proyecto: siempre digo que un gestor tiene como mayor prioridad «facilitar» el trabajo que realizan los miembros del equipo, como bien hemos analizado a lo largo de *El Libro Negro del Programador*. Navegamos en un mismo barco, de modo que debemos buscar buenos gestores (jefes) que sepan realizar su trabajo correctamente: debemos huir del «jefe-látigo» que piensa que escribir líneas de código requiere de las mismas habilidades que fabricar tornillos; precisamente, «la fábrica de tornillos» es una metáfora que he usado a menudo para referirme a compañías (grandes y pequeñas) que enfocan el software del mismo modo que una metalurgia: pueden producir productos tecnológicos, sí, pero con los mismos métodos… Se echa en falta una cultura de trabajo completamente distinta.

Cuando terminamos nuestra formación académica nos sentimos tranquilos y aliviados de haber superado por fin esos exámenes, asignaturas que se nos resistían y la cara de algún profesor al que difícilmente volveremos a ver; sin embargo, la maratón de fondo comienza en ese preciso momento: todo ese tiempo de aprendizaje da paso a una formación continua en la que a lo largo de años cambiaremos de paradigmas en varias ocasiones. Hemos visto cómo ciertas habilidades sociales y organizativas influyen poderosamente en el resultado de nuestro trabajo.

Siendo así las cosas, veo cómo la gran mayoría de programadores se ciñen exclusivamente a lo que se les exige en el trabajo que les paga la nómina a fin de mes: ¿podemos llegar a ser desarrolladores de software profesionales sin conocer otras tecnologías, sin leer un libro al mes, sin asistir a seminarios, sin aprender de aquellos que exponen sus trabajos altruistamente en Internet? Rotundamente no, de ningún modo.

¿En qué equipo estás? ¿En el que avanza a empujones con lo que te exigen de ocho a cinco o con el de las mentes inquietas que señalan artículos

en entradas de blogs para leerlos en su tiempo libre los fines de semana? Si estás en el primer grupo, siento decirte que estarás fuera del mercado antes que tarde y te tocará reciclarte dramáticamente en algún momento.

Los paradigmas de trabajo cambian: a veces no nos damos cuenta de que nosotros mismos podemos provocar esa misma tecnología disruptiva que genere un nuevo paradigma. Se me viene a la mente node.js[11] y su autor Ryan Dahl; gentes como Ryan, con sus brillantes ideas y proyectos, son las que a menudo hacen avanzar nuestro sector, y este, precisamente, las tecnologías de la información, están cambiando la economía a escala global.

Vivimos en la era del emprendimiento: la globalización (o «superglobalización» que diría Trías de Bes en su libro «El Gran Cambio») se lleva las horas de trabajo de manufacturas a Asia, por poner un ejemplo, de modo que en los países «desarrollados» o innovamos, emprendemos y nos ponemos las pilas, o nuestros trabajos serán exportados igualmente. Es una cuestión de costes, aritmética sencilla que pone al alcance de cualquiera la externalización de casi cualquier servicio (gracias, paradójicamente, a la misma tecnología a la que nos dedicamos).

Sin embargo el vaso está más lleno que vacío: las oportunidades, negocios, nuevas profesiones, nuevas formas de hacer las cosas, nuevos servicios están ahí al alcance de quien comience a pensar «por sí mismo» y, lo mejor de todo, todo este cambio de paradigma económico tiene como base nada más y nada menos que ¡el mismo software!, directamente o indirectamente.

Un programador tiene más oportunidades que nadie de innovar y emprender en este mundo tecnológico precisamente porque conoce las herramientas y los ladrillos con los que se está creando esta nueva arquitectura económica mundial. ¿Qué hacemos mientras tanto? Nos podemos beneficiar de esta nueva ola ejecutando los proyectos «de otros» en el mejor de los casos o bien podemos subirnos directamente a ella. Los átomos de la sociedad de la información son cientos de millones de líneas de código escritos por desarrolladores de software que, como nosotros, hacen que todo lo demás funcione.

El Libro Negro del Programador te ha mostrado las habilidades que debe tener todo buen desarrollador profesional (curiosamente muchas no técnicas)

y al mismo tiempo los errores que todos hemos cometido en algún momento y que han lastrado la calidad de los proyectos o nuestra propia evolución laboral y profesional. Confío en que tras su lectura seas aún mejor programador y sepas enfocar y dirigir mejor tus próximos pasos como desarrollador de software altamente productivo.

Mejorar en nuestra profesión, en el proyecto en el que trabajamos, en todo, es prácticamente una cuestión de hábitos y, como tales, se pueden aprender. Durante los capítulos de este libro he apuntado los hábitos más relevantes que debe incorporar cualquier desarrollador para escribir software de calidad y, por tanto, tener una carrera de éxito.

Gracias y nos seguimos viendo en www.ellibronegrodelprogramador.com

Sevilla, a 20 de febrero de 2014

El test del desarrollador de software altamente productivo

El espíritu de *El Libro Negro del Programador* es el de indicar qué impide por lo general producir software de calidad, incidiendo en aspectos que no son necesariamente técnicos pero especialmente relevantes para la buena marcha de un proyecto, sea este personal o en el contexto de una compañía. Un buen proyecto lo es también porque se ha realizado productivamente, sabiendo aprovechar las buenas prácticas, en su mayoría fáciles de poner en marcha, e incorporando los buenos hábitos de un buen desarrollador en el día a día.

A continuación se recogen en forma de preguntas las partes esenciales que se han discutido y analizado profusamente a lo largo de los capítulos anteriores. Con las respuestas a las siguientes cuestiones un desarrollador de software y un equipo de trabajo detectará todas aquellas áreas mejorables en su actividad diaria o bien pondrán en cuestión lagunas que hay que cubrir y solucionar.

Algunos tests pueden parecer obviedades para equipos de trabajo más avanzados; otros no lo son tanto pero los considero igualmente importantes. El conjunto, al final, es lo que cuenta.

1. ¿Está el código que generamos suficientemente respaldado por pruebas?
2. ¿Tiene el equipo en el que trabajamos una suficiente cultura de creación de pruebas?
3. ¿Dedicamos conscientemente algún tiempo a refactorizar?, esto es, ¿nos planteamos frecuentemente las preguntas de si algo se puede mejorar o simplificar?

4. ¿Vamos dejando en el código comentarios tipo «*to do:…*» que finalmente nunca se llegan a completar?
5. ¿Buscamos en nuestro día a día cómo aplicar principios S.O.L.I.D, KISS, etc.?
6. ¿Trabaja el equipo de desarrollo con la suficiente tranquilidad en un ambiente cordial y creativo?
7. ¿Existe suficiente cordialidad entre los miembros del equipo para cooperar en el proyecto?
8. ¿Tienen los proyectos en los que trabajamos que estar finalizados «para antes de ayer»?, es decir, ¿se trabaja siempre con prisas?
9. ¿Buscamos trabajar de cerca con la gente que mejor sabe cooperar y trabajar en equipo?
10. ¿Existen individualismos en el equipo o personas para las que les resulta difícil trabajar en grupo?
11. ¿Fomentan los managers o gestores el buen clima en el equipo de trabajo y ponen todos los medios para que este avance correctamente?
12. ¿Aplicamos intencionadamente principios de diseño cuando programamos?
13. ¿Tendemos a simplificar algo que ya funciona habitualmente?
14. ¿Sentimos la necesidad de llenar de comentarios internos las piezas de código que escribimos?
15. Cuando alguien retoma nuestro trabajo, ¿tiene que estar preguntándonos continuamente por detalles que no termina de entender?
16. ¿Buscamos la maestría en aquellas tecnologías que creemos que conocemos suficientemente?
17. ¿Aplicamos conscientemente las tácticas descritas en el libro de Martin Fowler sobre refactorings?
18. ¿Evaluamos correctamente las librerías y componentes externos que se usan en nuestro proyecto y tenemos en cuenta su grado de evolución, madurez, comunidad de usuarios, etc.?
19. ¿Aislamos adecuadamente las librerías y componentes externos en nuestra aplicación para no depender de ellos excesivamente?
20. ¿Somos suficientemente conscientes de que al proyecto en el que

trabajamos se le pedirán más cambios y que estos estarán en proporción a su éxito?

21. ¿Tenemos la tendencia de usar siempre lo último «porque sí» sin considerar si su uso es adecuado (y seguro) en nuestro proyecto?
22. ¿Preferimos usar tecnologías muy maduras antes que tecnologías incipientes y de rápida evolución?
23. ¿Consideramos el riesgo de usar módulos, librerías o componentes relativamente nuevos en nuestros proyectos?
24. ¿Estamos dispuestos a modificar profundamente una pieza de código en la que estuvimos trabajando mucho tiempo?
25. ¿Intentamos mantener a toda cosa un trozo de código sabiendo que se puede mejorar?
26. ¿Somos honestos con nosotros mismos cuando llegamos a la conclusión de que es mejor comenzar algo desde cero que forzar de la manera que sea la incorporación de nuevas características?
27. ¿Permitimos incluir en el equipo más gente en momentos de crisis cuando se acercan las fechas de entrega y se ve que de ningún modo se va a completar el trabajo?
28. ¿Advertimos a nuestros responsables de la falta de recursos para ejecutar con calidad y éxito el proyecto en el que trabajamos?
29. Si somos responsables de equipos, ¿tenemos claro que cuando un equipo falla es nuestra responsabilidad?
30. ¿Sufrimos demasiadas reuniones improvisadas?
31. ¿Las reuniones terminan durando mucho más del tiempo establecido y se discuten más temas de los incluidos en la agenda?
32. ¿Hacemos siempre un trabajo que previamente ha sido planificado por el gestor del proyecto?
33. ¿Se pasa por alto la metodología o las buenas prácticas en momentos de especial estrés o crisis por llegar a las fechas previstas?
34. ¿Cambia el gestor de proyecto de criterio continuamente?
35. ¿Tenemos todos los medios necesarios para hacer un buen trabajo?
36. ¿Comenzamos un nuevo proyecto o fase de desarrollo resolviendo las partes más complicadas o que más nos inquietan?

37. ¿Aplicamos continuamente refactorings al trabajo realizado o sólo cuando nos acordamos?
38. ¿Trabajamos siempre con la idea de la calidad en mente, eso es, queriendo hacer siempre el mejor trabajo?
39. ¿Dominamos suficientemente bien las tecnologías que se emplean en el proyecto?
40. ¿Nos ceñimos fielmente a la metodología que se usa desde el principio hasta al final del proyecto?
41. ¿Abandonamos las buenas prácticas metodológicas cuando sentimos presión?
42. ¿Trabajamos en equipos con talentos descompensados, esto es, unos muy buenos y otros muy «malos»?
43. ¿Se trabaja estableciendo una arquitectura general rígida al principio del proyecto cuando aún no están claros todos los requisitos?
44. ¿Es la arquitectura general del proyecto suficientemente flexible para poder modificarla durante el desarrollo del mismo o es extraordinariamente rígida?
45. ¿Está sólidamente establecida la metodología que se ha decidido seguir o se ignora a primeras de cambio?
46. ¿Se percibe claramente la rentabilidad que se extrae de seguir ¡correctamente una metodología?
47. ¿Se relegan siempre para el final las tareas más aburridas o rutinarias?
48. ¿Nos preocupamos lo suficiente en que nuestro producto software esté bien hecho y funcione y además que lo aparente?
49. ¿Ponemos suficiente énfasis en el diseño de interfaces de usuario ágiles, intuitivas, amigables y elegantes?
50. ¿Nos ocurre a menudo que diseñamos interfaces de usuario pensando más en los propios desarrolladores que en los usuarios finales?
51. ¿Conocemos muy superficialmente multitud de tecnologías pero muy pocas en verdadera profundidad?
52. ¿Nos centramos más en conocer algo de muchas tecnologías que en resolver problemas reales con nuestro software?
53. ¿Sentimos la necesidad de usar lo último de lo último en un nuevo

desarrollo «porque sí»?

54. ¿Indicamos en nuestro currículum los problemas que hemos resuelto con nuestro software o bien una retahíla de tecnologías que conocemos superficialmente?
55. ¿Tenemos el trabajo planificado con tiempo de antelación?
56. ¿Es corriente llegar a la oficina sin saber exactamente qué tenemos que hacer?
57. ¿Nos marcan las tareas que debemos realizar sobre la marcha?
58. ¿Sufrimos continuamente interrupciones durante nuestro trabajo?
59. ¿Contamos con los recursos y medios necesarios para realizar correctamente nuestro trabajo?
60. ¿Se realizan horas extra con frecuencia para completar las tareas?
61. ¿Percibimos que nuestro gestor o manager se preocupa lo suficiente para que trabajemos concentrados la mayor parte del tiempo en nuestras tareas?
62. ¿Intentamos reutilizar frameworks y librerías suficientemente maduras en el desarrollo de nuestros proyectos?
63. ¿Nos preocupamos de que las partes funcionales de la aplicación estén suficientemente desacopladas entre ellas?
64. ¿Nos gusta reinventar la rueda en lugar de usar librerías que ya hacen lo que necesitamos porque creemos que lo haremos mejor?
65. ¿Nos esforzamos siempre es escribir código que sea fácil de depurar y corregir?
66. ¿Son nuestras soluciones imposibles de depurar y de corregir cuando son lanzadas a producción?
67. ¿Nos mantenemos muchísimo tiempo trabajando en el mismo proyecto y empleando las mismas tecnologías que hace algunos años?
68. ¿Rotamos periódicamente de proyectos?
69. ¿Detectamos en el equipo o departamento a quienes mantienen en secreto cierto conocimiento e información sobre alguna solución?
70. ¿Somos altamente imprescindibles en una tarea muy específica y crítica que realizamos en la compañía?
71. ¿Tenemos una actitud de compartir con los demás todo lo que

realizamos o aquello que conocemos mejor?

72. ¿Revisamos proyectos de otros desarrolladores?
73. ¿Leemos con frecuencia artículos de desarrollo sobre las tecnologías que usamos habitualmente?
74. ¿Participamos en proyectos heterogéneos?
75. ¿Ponemos en marcha algún tipo de gestión de la configuración antes de comenzar un nuevo proyecto?
76. ¿Se sigue fielmente la gestión de la configuración definida a lo largo de todo el tiempo de desarrollo del proyecto?
77. ¿Practicamos integración continua en grupos de trabajo de más de dos desarrolladores?
78. ¿Son conscientes todos los miembros del equipo de la importancia de trabajar fielmente con los procedimientos de gestión de la configuración e integración continua definidos?

Bibliografía

Puedo considerar muchos libros como base para haber escrito *El Libro Negro del Programador*; podrás ver que muchos no tienen nada que ver con el software, pero sí con formas de «pensar» diferentes para afrontar mejor tu trabajo, innovar, emprender y seguir aquello que realmente quieres en tu vida.

Algunos de ellos son libros para mí de referencia que leo a menudo y de los que con cada nueva lectura aprendo una nueva gema de sabiduría que aplicar en mi día a día.

ANDREW HUNT / DAVID THOMAS: "*The Pragmatic Programmer*".

CHAD FOWLER: "*The Passionate Programmer: Creating a Remarkable Career in Software Development*".

CHRIS GUILLEBEAU: "*100€ startup*".

DAVID ALLEN: "*Organízate con Eficacia*".

ERIC RIES: "*Lean Startup: How Relentless Change Creates Radically Successfull Businesses*".

FERNANDO TRÍAS de BES: "*El Libro Negro del Emprendedor*".

FERNANDO TRÍAS de BES: "*El Gran Cambio*".

JOHANNA ROTHMAN / ESTHER DERBY: "*Behind Closed Doors: Secrets of Great Management*".

JON BENTLEY: "*Programming Pearls*".

JONATHAN RASMUSSON: "*The Agile Samurai: How Agile Masters*

Deliver Great Software”.

MARTIN FOWLER: “*Refactoring: improving the Design of Existing Code*”.

MAX KANAT-ALEXANDER: “*Code Simplicity*”.

RAIMON SAMSÓN: “*El Código del Dinero*”.

ROBERT C. MARTIN: “*Clean Code: A handbook of Agile Software Craftsmanship*”.

ROBERT C. MARTIN: “*The Clean Coder: A Code of Conduct for Professional Programmers*”.

ROBERT C. MARTIN / MICAH MARTIN: “*Agile Principles, Patterns and Practices in C#*”.

SERGIO FERNÁNDEZ: “*Vivir sin Jefe*”.

SERGIO FERNÁNDEZ: “*Vivir sin Miedos*”.

STEPHEN R. COVEY: “*Los 7 Hábitos de la Gente Altamente Efectiva*”.

STEVE KRUG: “*Don’t Make me Think: A Common Sense Approach to Web Usability*”.

STEVEN MCCONNELL: “*Code Complete: A Practical Handbook of Software Construction*”.

TIMOTHY FERRISS: “*La Semana Laboral de 4 Horas*”.





RAFAEL GÓMEZ BLANES es Ingeniero Informático por la Universidad de Sevilla (España). Infoemprendedor, ha trabajado en proyectos software internacionales relacionados con el sector eléctrico. Desarrollador profesional desde el año 1998, es experto en *clean code* y todas aquellas prácticas metodológicas que incrementan la productividad, mejorando la calidad del software generado. Evangelista de software ágil, dirige actualmente un equipo de desarrollo en una compañía de ingeniería realizando productos para la gestión de *smart meters* y su despliegue en la nube en modo SaaS (*software as a service*).

Notas

[1] *Continuous delivery*

http://en.wikipedia.org/wiki/Continuous_delivery. <<

[2] Manifiesto ágil

http://agilemanifesto.org/principles.html. <<

[3] Obsolescencia programada

http://es.wikipedia.org/wiki/Obsolescencia_programada. <<

[4] Inyección de dependencias

http://es.wikipedia.org/wiki/Inyección_de_dependencias. <<

[5] *Code Simplicity* de Max Kanat-Alexander. <<

[6] *Software rot*

http://en.wikipedia.org/wiki/Software_rot. <<

[7] *Stand-up meetings*: reuniones muy cortas para indicar «rápidamente» el estado de cosas.

http://en.wikipedia.org/wiki/Stand-up_meeting. <<

[8] *Done means done*: principio del desarrollo ágil por el que una característica debe cerrase completamente antes de pasar a la siguiente, lo que implica haber sido desarrollada en todos sus detalles, probada y aceptada como terminada. <<

[9] *The Passionate Programmer* de Chad Fowler. <<

[10] *Cowboy Coding*: forma de codificar sin estructura metodológica alguna, útil para experimentación y pruebas.

http://en.wikipedia.org/wiki/Cowboy_coding. <<

[11] Node js

http://es.wikipedia.org/wiki/Nodejs. <<